Carl-Marcus Weitz

RTOS PEARL

Integriertes Echtzeit-Multitasking-Programmiersystem

Eine kleine Einführung in PEARL-UH

HEISE

Carl-Marcus Weitz

RTOS PEARL

Integriertes
Echtzeit-
Multitasking-
Programmiersystem

Eine kleine
Einführung
in PEARL-UH

HEISE

Impressum
© by Verlag Heinz Heise GmbH, Hannover
Alle Rechte vorbehalten.
Redaktion: Christian Persson
Umschlaggestaltung und Layout: Wolfgang Ulber
Titelfoto: Lutz Reinecke, Hannover
ISBN 3-88229-050-1

Inhalt

Die ersten Schritte 1
Modul-Aufbau, Spezifizieren und Deklarieren, Die erste Task

c't-Userport unter PEARL 2

Ein kleines Adreßprogramm 3
Compiler, Ein-/Ausgabe, Datentypen, Prozeduren, CASE-Auswahl

Editieren beim Formatieren ...? 9

Bedienbefehle timer-gesteuert 9

Maus, Grafik, Interrupts 16
Ereignisgesteuerte Einplanung, Task-Synchronisierung

Assembler-Programmierung in PEARL-Umgebung 24
Hyperprozessor, Assembler-Prozeduren und -Tasks

RTOS-Pe(a)rlen 31
Compiler-Optionen, Tracing, Speicherverwaltung

Wir machen Musik 40
Zugriff im Supervisor-Modus, Dispatcher-OFF

Benchmarke unter RTOS 43

Programme in EPROMs 44
EPROMs brennen „nebenbei"

Software-Upgrading 52
Pipes, Time-Sharing, neue Einbaufunktionen, REFs

Carl-Marcus Weitz

Programmieren in PEARL

Die ersten Schritte

In den folgenden Kapiteln geht es um die Programmiersprache, die in idealer Weise die Leistungsfähigkeit von RTOS nutzbar macht. Ein Betriebssystem ist schließlich immer nur Grundlage der Programme, die unter ihm laufen. Daß es hier als Programmiersprache PEARL vorgestellt wird, ist weniger verwunderlich, wenn man sich die Geschichte des RTOS ansieht: es wurde ja entwickelt, um für PEARL eine geeignete Betriebssystemumgebung auf kleineren Rechnern zu schaffen.

Der Sprachumfang von PEARL ist in zwei Stufen nach DIN genormt. Basic-PEARL besitzt einen mit anderen modernen Hochsprachen für Mikrocomputer vergleichbaren Sprachumfang, Full-PEARL ist dagegen so umfangreich wie etwa ALGOL 68 und eigentlich nur für den Einsatz auf Großrechenanlagen geeignet. Der Sprachumfang des Compilers im RTOS-UH/PEARL-Paket liegt, grob ausgedrückt, irgendwo zwischen den beiden Leistungsstufen. Wenige kleine Abweichungen von Basic-PEARL bestehen allerdings auch; auf diese wird im folgenden noch hingewiesen.

Wenn man einer neuen Programmiersprache begegnet, fängt man häufig damit an, daß man irgendeinen kleinen Text auf den Bildschirm zu schreiben versucht. Und genau dieses tut das kleine Beispielprogramm TEST1. Manchen mag der relativ große Kopf des Programmes erschrecken für eine Aufgabe, die in BASIC mit dem Einzeiler

10 PRINT 'HELLO WORLD'

zu erledigen wäre. Doch dieser Wasserkopf ist notwendig, um die Portabilität der Programme zu erreichen. Und zum Glück wächst der Umfang des Kopfes bei zunehmender Länge des Hauptprogramms nicht zwangsläufig mit.

Doch nun zu dem ersten Beispielprogramm. Die Zeilen 1 bis 5 sind Kommentare. Kommentare dürfen überall dort stehen, wo ein Leerzeichen stehen darf, also auch mitten in Anweisungszeilen (was man aus Gründen der Übersicht allerdings wohl seltener machen wird). Sie werden vom Compiler zumeist ignoriert. Zumeist deshalb, weil man, wie bei einigen Pascal-Compilern, in Kommentarzeilen Compiler-Optionen an- und abschalten kann. Doch das gehört schon zu den höheren Programmierweihen und wird später erläutert. Bis dahin nehmen wir an, daß alles, was zwischen den Endsymbolen '/*' und '*/' steht, nur für den Programmie-

```
*UH-PEARL-9.9                                   Copyright:Ge/IRT-Uni-Han,4.1986
=     1 /*****************************************************************/
=     2 /*                                                               */
=     3 /*   PROGRAMMIEREN IN PEARL     Beispiel 1                       */
=     4 /*                                                               */
=     5 /*****************************************************************/
=     6
=     7 MODULE TEST1;
=     8
=     9 SYSTEM;
=    10     TERM: A1;
=    11
=    12 PROBLEM;
=    13     SPC TERM DATION INOUT ALPHIC CONTROL(ALL);
=    14
=    15 BEISP1: TASK;
=    16     PUT 'HELLO WORLD' TO TERM;
=    17 END;    /* OF TASK BEISP1 */
=    18
=    19 MODEND;
```

Ein winziges Übungsprogramm zum ersten Einstieg in PEARL.

rer bestimmt ist, damit er sich in seinen Programmen zurechtfindet (und andere Programmierer vielleicht auch einmal). Denn natürlich könnte man mit einiger Anstrengung auch in PEARL unlesbare Programme schreiben, die man schon nach einigen Tagen selbst nicht mehr versteht.

Schlüsselwort MODULE

Mit dem Schlüsselwort 'MODULE' in Zeile 7 betreten wir eine eigene Welt, die erst mit dem Gegenstück 'MODEND' in Zeile 19 wieder aufhört. In dieser Welt kennen sich die Tasks, man kann gemeinsame Variablen haben und gemeinsam Prozeduren und Funktionen benutzen. Zu beachten ist, daß ein solches Objekt, bevor es angesprochen werden kann, im Programmtext definiert worden sein muß, da der Compiler ja in einem einzigen(!) Durchlauf den Programmcode erzeugt und natürlich nicht hellsehen kann, was für ein Datentyp sich hinter einer Variablen verbirgt oder welche Parameter eine Prozedur benötigt.

Hinter 'MODULE' kann, getrennt durch ein Leerzeichen, ein Name folgen, der 24 Buchstaben lang sein darf. Unter diesem Namen findet man das geladene Modul dann später in der Speicherverwaltung wieder. Von der Möglichkeit, ein Modul zu benennen, sollte man Gebrauch machen, denn man will ja eventuell das Modul irgendwann wieder aus dem Speicher entfernen. Wird nämlich kein Modulname angegeben, so nennt RTOS das Modul einfach 'UNLBLD' (unlabeled), und man weiß dann bei mehreren Modulen nicht, welches man löscht. Denn mit 'UNLOAD Modulname' löscht man immer das erste Modul dieses Namens in der Speicherverwaltung (deshalb sollte man auch übereinstimmende Modulnamen vermeiden).

Die Welt, in die wir soeben eingetreten sind, fängt (wie so manche andere Welt auch) mit Verwaltung an. Doch zum Glück ist diese Verwaltung noch überschaubar. Unser Modul möchte nämlich wissen, was es außerhalb seiner Welt gibt, wie das Rechnersystem aussieht, auf dem es läuft, wie die Ansprechpartner für Ein- und Ausgabe heißen. Dies teilen wir hinter

c't-Userport unter PEARL

Das RTOS-UH/PEARL-Paket für die Atari-ST-Serie unterstützt den c't-Userport, der je 16 Leitungen für die Ein- und Ausgabe von Daten zur Verfügung stellt (siehe c't 3/86). Wer den Beitrag gelesen hat, wird sich erinnern, daß für die Ausgabe über den ROM-Port des ST ein Trick verwendet werden mußte, weil der Rechner keine Schreiboperationen auf die betreffenden Adressen gestattet: Die Datenausgabe erfolgt durch Lesen bestimmter Adressen.

Ein Hochsprachen-Compiler macht solche Tricks normalerweise natürlich nicht mit, jedenfalls nicht mit einer schlüssigen sprachlichen Konstruktion. Anders der UH-Compiler: er bot in einer Variante für das Force-Profikit schon die Möglichkeit, auf einen speziellen Unterbus (P-Bus) zuzugreifen. Dieses Feature wurde beim Atari ST zur Unterstützung des Userports 'umgebogen'. Um die 16 Ausgabeleitungen als Datenstation nutzen zu können, kann man nun folgende Konstruktion verwenden:

```
SYSTEM;
PORT: BU(0203)->;
...........

PROBLEM;
SPC PORT DATION OUT BASIC;
...........

AUSGAB: TASK;
DCL DATA BIT(16);

SEND DATA TO PORT;

...........
```

Die Eingabeports und übrigen Register des ST-Userports können ganz einfach als Datenstationen mit dem Namen BU und ihrer jeweiligen Adresse verwendet werden, also beispielsweise der Eingabeport für direkten Zugriff unter BU(02FAFFF8), wobei das erste Byte des Ausdrucks in Klammern die Zugriffsart beschreibt (siehe Handbuch).

dem Schlüsselwort 'SYSTEM' dem Modul mit. In unserem Beispiel besteht die Außenwelt für das Modul nur aus dem Terminal, dem hier der Name 'TERM' gegeben wird. Dieser Name ist frei wählbar, man kann sogar als modul-internen Namen den Namen der zugehörigen Datenstation, A1 (siehe Abschnitt A-II), nehmen. Unter diesem Namen wird von nun an das Terminal angesprochen.

Mancher wird sich jetzt natürlich fragen, wozu so ein Systemteil eigentlich gut ist. Ein Punkt ist die schon oben erwähnte Portabilität: wenn ein Programm auf einem anderem Rechnersystem laufen soll, muß man im Idealfall nur den Systemteil ändern. Aber auch für den Programmierer, der nur für ein Rechnersystem schreibt, kann er nützlich sein: man braucht nämlich nur den Namen der Datenstation im Systemteil ändern, und schon gehen alle Daten, die vorher zum Beispiel auf den Bildschirm gingen, an den Drucker. Doch das wird weiter unten noch genauer beschrieben.

Der zweite Teil des Modules wird in Zeile 12 eingeleitet durch 'PROBLEM'. Formal gehört alles folgende zum Problemteil. In dem Problemteil steht alles das, was nicht von dem Betriebssystem abhängt, auf dem das Programm laufen soll. Wenn jemand ein Programm betriebssystemunabhängig schreibt, kann er auf den Systemteil verzichten und gleich mit dem Problemteil beginnen. Um eine lauffähige Version zu erhalten, muß dann nur noch für das vorgesehene Betriebssystem ein Systemteil eingefügt werden.

Nachdem den Datenstationen des Betriebssystems nun logische Namen für das Modul zugeordnet sind, müssen diese noch genauer bezüglich ihrer Eigenschaften beschrieben werden. Einem Terminal beispielsweise sollen Zahlen, die rechnerintern binär codiert sind, in den ASCII-Code umgewandelt zu-

Spezifizieren und deklarieren

gesandt werden. Von einem Drucker darf beispielsweise keine Eingabe erwartet werden. Für jede im Systemteil aufgeführte Datenstation muß deshalb im 'PROBLEM'-Teil eine Beschreibung, eingeleitet durch 'SPC' oder 'SPECIFY', stehen. Die Möglichkeiten hier aufzuzählen, wie diese Beschreibung aussehen kann, würde zu weit führen, da PEARL für die unterschiedlichsten Datenstationen Beschreibungsmöglichkeiten vorsieht, vom Meßfühler bis zur Datenfernübertragungsstation. Deshalb hier nur das, was zur Beschreibung unseres Terminals notwendig ist:

Hinter dem 'SPC' steht der Name, den wir der Datenstation zugewiesen hatten, hier also 'TERM', gefolgt von dem Schlüsselwort 'DATION', damit der Compiler weiß, daß es sich hier um eine Datenstationsbeschreibung handelt. 'INOUT' bedeutet, daß sowohl Eingabe (über die Tastatur) als auch Ausgabe (über den Bildschirm) möglich ist. 'ALPHIC' besagt, daß Zahlen und Bitmuster in die entsprechenden ASCII-Codes umgewandelt werden sollen, beziehungsweise umgekehrt bei der Eingabe, daß ASCII-Codes in Bitmuster und Zahlen umzusetzen sind. Mit 'CONTROL (ALL)' wird schließlich gesagt, daß diese Datenstation Control-Zeichen wie CR (Carriage Return, Ausgabe in erster Spalte fortsetzen) und LF (Line Feed, Zeilenvorschub) versteht, die zur formatierten Ausgabe von Text auf Bildschirm oder Drucker notwendig sind.

Die erste Task

So, die Verwaltung wäre geschafft, unsere erste Task beginnt, und zwar mit ihrem Namen und, durch Doppelpunkt getrennt, der Information, daß es sich tatsächlich um eine solche handelt, indem es dort als Schlüsselwort steht: 'TASK'. Durch Eingabe dieses Namens wird dann später die Task gestartet. Beachtenswert auch hier, daß Namen nicht doppelt auftauchen sollten, auch nicht in verschiedenen Modulen.

Und damit wären wir bei der eigentlichen Aufgabe angelangt, für die das ganze Modul geschrieben wurde: in Zeile 16 wird der Text 'HELLO WORLD' auf den Bildschirm geschrieben. Der Befehl 'PUT', der dieses bewirkt, wird gefolgt von den Operanden, hier nur dem String 'HELLO WORLD'. Sie werden später noch sehen, daß hier auch mehrere Operanden verschiedener Art stehen dürfen. Mit 'TO' ist die Operandenliste dann abgeschlossen, und als nächstes folgt Name der Datenstation als Zielangabe für die Ausgabe. Die Task wird durch 'END' abgeschlossen, und mit 'MODEND', das man nicht vergessen darf, ist das Programm fertig.

Wer dieses Programm als ED-File ED.SI eingegeben hat, braucht jetzt nur noch mit 'ESC X' den Editor wieder zu verlassen, 'P', 'LOAD' und 'BEISP1' einzugeben und erhält (so er sich nicht vertippt und der Compiler nicht deshalb angemahnt hat) die freundliche Begrüßung auf den Bildschirm.

Mehrere Tasks gleichzeitig

'Das hätte man aber einfacher haben können!' wird jetzt so mancher mit Blick auf den BASIC-Einzeiler oder andere Programmiersprachen denken. Aber dieses erste Programm war ja auch nicht gedacht, um die Leistungsfähigkeit von PEARL zu demonstrieren, sondern um das Gerüst vorzustellen, das jetzt weiter ausgefüllt werden soll. Und zwar gleich mit mehreren Tasks, um zu demonstrieren, welche Möglichkeiten sich dem Programmierer durch 'Multitasking' unter RTOS eröffnen.

Dazu soll eine Task ('ACTION') alle 50 Millisekunden den Wert einer Variablen, ihr Quadrat und den Sinus auf den Bildschirm ausgeben und für den nächsten Durchlauf ihren Wert um 1 erhöhen. Bei dieser Aufgabe soll sie eine andere Task ('START') beobachten und in gleichmäßigen Abständen den Wert, den die erste Task gerade bearbeitet, speichern, um ihn später anzuzeigen. Diese Task soll auch die Ein- und Ausplanung der ersten Task vornehmen sowie eine dritte Task starten, die in einer Endlosschleife ihre Runden dreht, dabei eine Variable hochzählend, die als Maß für die Anzahl der gedrehten Runden steht. Spätestens bei diesem Programm zeigt sich, daß PEARL eine etwas andere Sprache ist als die meisten anderen. Denn wenn ein Programm in einer anderen Programmiersprache etwas wie

10 GOTO 10

enthält, hilft meist nur noch 'Break', 'Reset' oder gar Ausschalten des Rechners. Sinnvoll einsetzen läßt es sich jedenfalls nicht. Ganz anders hier: die Task REST soll ja gar nicht von allein zur Ruhe kommen, und dafür ist eine Endlosschleife genau das richtige Mittel.

Eine Endlosschleife besteht aus dem Schlüsselwort 'REPEAT' (Zeile 26) und einem 'END' (Zeile 28). Jedesmal, wenn eine Task auf so ein Schleifen-'END' läuft, beginnt sie wieder bei dem dazugehörigen 'REPEAT'. Bei den normalen Schleifen mit Ende steht dann vor dem 'REPEAT' die Abbruchbedingung (WHILE Bedingung REPEAT ... END;), ein Schleifenindex (FOR Zählvariable FROM Anfangswert BY Schrittweite TO Endwert REPEAT ... END;) oder auch, was auch nicht ganz gewöhnlich ist, beides (FOR Zählvariable FROM Anfangswert BY Schrittweite TO Endwert WHILE Bedingung REPEAT ... END;). Letzteres ist eine erlaubte Konstruktion in PEARL, durch die man sich vor

```
*UH-PEARL-9.9                                  Copyright(c)IRT-Uni-Han,4.1986
=    1 /*****************************************************************/
=    2 /*                                                               */
=    3 /*   PROGRAMMIEREN IN PEARL     Beispiel 2                        */
=    4 /*                                                               */
=    5 /*****************************************************************cmm*/
=    6
=    7 MODULE TEST2;
=    8
=    9 SYSTEM;
=   10      TERM: A1;
=   11
=   12 PROBLEM;
=   13      SPC TERM DATION INOUT ALPHIC CONTROL(ALL);
=   14
=   15      DCL X FLOAT;                /* Rechenvariable */
=   16      DCL LEERLAUF FIXED(31);     /* Zaehlvariable fuer Restzeit */
=   17
=   18
=   19 ACTION: TASK PRIO 39;
=   20      PUT X, X * X, SIN(X) TO TERM BY F(5),F(8),E(12,2),SKIP;
=   21      X = X + 1.0;
=   22 END;    /* OF TASK ACTION */
=   23
=   24
=   25 REST: TASK PRIO 40;
=   26      REPEAT                      /* Endlosschleife */
=   27        LEERLAUF = LEERLAUF + 1;
=   28      END;                        /* Ende der Endlosschleife */
=   29 END;    /* OF TASK REST */
=   30
=   31
=   32 START: TASK PRIO 38;
=   33      DCL FELD(10) FLOAT;
=   34        X = 1.0;                  /* initialisiere Rechenvariable */
=   35        LEERLAUF = 0;             /* initialisiere Leerlaufindex */
=   36        ALL 0.05 SEC ACTIVATE ACTION; /* plane ACTION ein */
=   37        ACTIVATE REST;            /* starte Task REST */
=   38
=   39        FOR I FROM 1 TO 10 REPEAT   /* registriere, wie weit Task */
=   40          FELD(I) = X;              /*  ACTION inzwischen ist     */
=   41          AFTER 0.2 SEC RESUME;     /* warte ein wenig */
=   42        END;
=   43
=   44        PREVENT ACTION;           /* nimm ACTION aus zyklischer Einplanung */
=   45        TERMINATE REST;           /* erloese REST aus Endlosschleife */
=   46
=   47        PUT TO TERM BY (2)SKIP;     /* gib 2 Leerzeilen aus */
=   48        PUT 'Rechenvariable nach jeweils 0.2 Sekunden:'
=   49         TO TERM BY A,SKIP;
=   50        FOR I FROM 1 TO 10 REPEAT   /* gib registrierte Werte aus */
=   51          PUT I/5, FELD(I) TO TERM BY F(6,1),F(8),SKIP;
=   52        END;
=   53        PUT 'Leerlaufindex: ',LEERLAUF TO TERM BY SKIP,A,F(8),SKIP;
=   54 END;    /* OF TASK START */
=   55
=   56
=   57 MODEND;
```

In diesem Beispielprogramm arbeiten bereits mehrere Tasks zusammen.

ungewollten Endlosschleifen schützen kann, indem man außer einer Abbruchbedingung, von der man nicht sicher weiß, ob sie eintritt, noch eine Schleifenzählung einrichtet und bei Fernbleiben der Abbruchbedingung dann eben spätestens nach 10000 Schleifendurchläufen oder so ähnlich abbricht.

Steht nun aber vor dem 'REPEAT' nichts, so ist die Schleifenbedingung immer erfüllt, und wir haben unsere Endlosschleife. Ist aber eine Schleifenbedingung angegeben und nicht erfüllt oder eine Schleifenzählung abgelaufen, so wird die Task hinter dem Schleifen-'END' fortgesetzt. Beispiele für Schleifen mit Zählvariable finden sich in den Zeilen 39 - 42 und 50 - 52. Die Zählvariable 'I' muß nicht, wie etwa in C, ausdrücklich deklariert werden, da sie immer vom Typ 'FIXED', also ganzzahlig sein muß; sie kann dann aber nur innerhalb der Schleife benutzt werden. Die Schrittweite wird automatisch mit 1 angenommen, falls nicht anders angegeben.

Doch schön der Reihe nach: Was passiert eigentlich, wenn man, nachdem man das Modul fehlerfrei abgetippt, übersetzt und geladen hat, mit 'START' die Task selbigen Namens aktiviert? Im Quelltext fällt bei diesem Task-Namen der Zusatz 'PRIO 38' auf. Ähnliche Zusätze haben auch die anderen Tasks in dem Modul, nur mit anderen Zahlen. Hierdurch bekommen die Tasks eine Priorität zugewiesen. Wenn mehrere Tasks gleichzeitig laufen dürfen, muß ja irgendwie entschieden werden, welche den Prozessor zugeteilt bekommt, und das ist die mit der höchsten Priorität. Für unser Modul bedeutet das, daß, wenn alle Tasks laufen dürfen, zuerst START an der Reihe ist, bis sie nicht mehr darf, dann ACTION und ganz zuletzt REST. Wie sich dies auswirkt, wird gleich beschrieben.

Erst einmal haben wir in Zeile 33 eine Variablendeklaration, und zwar nicht nur eine einzelne Variable, sondern gleich ein ganzes Feld. Dieses Feld ist eindimensional und enthält als Elemente zehn 'FLOAT'ing-Point-Zahlen. Dieses Feld ist 'local', es kann also nur innerhalb der Task START darauf zugegriffen werden, außerhalb ist es unbekannt. Anders sieht es diesbezüglich mit 'X' in der nächsten Zeile aus. Dies ist eine 'globale' Variable, auf die alle Tasks im Modul zugreifen können. Deshalb muß sie auch außerhalb aller Tasks deklariert werden, was in Zeile 15 geschehen ist. Damit sie einen definierten Wert hat, wird ihr in Zeile 34 einer zugewiesen. Selbiges geschieht mit LEERLAUF, auch einer globalen Variablen, in Zeile 35.

In Zeile 36 wird nun ein Zug der Leistungsfähigkeit von PEARL sichtbar: hier weist nämlich die Task START das Betriebssystem an, alle 50 Millisekunden doch bitte die Task ACTION zu starten. Und sofort wird mit Zeile 37 die Task REST gestartet. Das bedeutet nun aber noch lange nicht, daß ACTION oder REST jetzt tatsächlich schon anfangen zu laufen. Davor hat das Betriebssystem die Prioritäten gesetzt, und die besagen, daß, obwohl ACTION und REST jetzt laufen dürfen, erstmal noch START weiterläuft, da sie die höchste Priorität hat. Also weiter in der Task START. Als nächstes kommt die schon weiter oben erwähnte Schleife. In dieser Schleife liest START den Wert von X in das Feld FELD ein. Das wäre aber schön langweilig, wenn sich nichts an dem Wert von X täte, und deshalb wird hier eine weitere Spezialität von PEARL eingesetzt: mit dem Befehl 'AFTER Zeitdauer RESUME' kann sich ein Programm selbst für die Zeitdauer abmelden, wenn es momentan nichts mehr zu tun hat, und das Betriebssystem bitten, es danach wieder aufzuwecken und weiterlaufen zu lassen.

Und damit ist endlich der Augenblick gekommen, in dem ACTION losläuft, die ja schon seit Zeile 36 sehnsüchtig darauf wartet. Aber ACTION ist ein Sprinter, schon nach etwa 30 Millisekunden ist sie fertig und damit der Weg frei für REST, sich mit atemberaubendem Tempo im Kreise zu drehen. Doch dies währt nicht lange, denn alle 50 Millisekunden sollte ja ACTION laufen, und da ACTION eine höhere Priorität hat als REST, ist für REST der Spaß nach längstens 20 Millisekunden vorbei, und sie muß wieder warten, bis ACTION fertig ist. So wechseln sich diese beiden Tasks ab, bis dem Betriebssystem einfällt, daß es ja START nach 200 Millisekunden wieder wecken sollte, und da START unter den Dreien die höchste Priorität hat, müssen ACTION und REST warten, bis START den inzwischen von ACTION fleißig inkrementierten Wert von X in FELD abgespeichert hat und das Spiel von neuem losgeht.

Jeder Spaß hat irgendwann ein Ende, und dieser Zeitpunkt ist für ACTION und REST gekommen, wenn START feststellt, daß das FELD voll ist. Dann verläßt sie nämlich ihre Schleife, läßt als erstes das Betriebssystem die regelmäßige Einplanung von ACTION mit 'PREVENT' zurücknehmen und befreit REST mit 'TERMINATE' aus ihrer Endlosschleife. Diese beiden Befehle haben die gleiche Wirkung wie die gleichnamigen Bedienerbe-

fehle; sie haben zur Folge, daß ACTION und REST wieder in den Status 'DORM' wie dormant oder schlafend verfallen.

Bevor wir uns jedoch weiter der Task START widmen, wollen wir uns noch ansehen, was START so beobachtet hat. Wenig verwunderlich ist, daß ACTION zwischen zwei Beobachtungen X um 4 hochgezählt hat, bei 50 Millisekunden Periodenzeit für ACTION und 200 Millisekunden Wartezeit für START entspricht das den Erwartungen. Erstaunlicher ist da wohl schon eher, daß die auf niedrigster Priorität laufende Task REST in den zwei Sekunden, in denen sie überhaupt lauffähig war, sich immerhin noch rund 95000mal im Kreise drehen konnte, obwohl in der ATARI-Implementation von RTOS ja eine Terminal-Emulation enthalten ist und der Prozessor somit noch etliche Pixel beim Scrollen über den Bildschirm zu schieben hatte.

Doch nun zurück zu der Task START, die nun auf dem Bildschirm anzeigt, was sie beobachtet hat. Dabei fällt als erstes in Zeile 47 auf, daß das 'TO' direkt auf das 'PUT' folgt. Diese Ausgabeanweisung dient nur dazu, zwei 'SKIP's, die Zeilenvorschübe bewirken, abzusetzen. Hinter 'BY' kann man in Ein- und Ausgabeanweisungen an Datenstationen vom Typ 'ALPHIC' Angaben machen, wie die Ein-/Ausgabe aussehen soll. In einer Ausgabeanweisung bedeutet ein 'SKIP', daß ein Zeilenvorschub gemacht werden soll. Für die auszugebenden Daten kann man angeben, wie lang (in Zeichen) die Ausgabe werden soll. So könnte man in unserem kleinen Beispiel hinter 'TO TERM' noch 'BY SKIP, A,SKIP;' anfügen, um vor und hinter dem Text je eine Leerzeile auszugeben. Das 'A' steht für die Ausgabe von Daten des Typs 'CHAR', die keiner Längenbeschränkung unterliegen.

Will man es sich mit dem Formatieren einfacher machen, so kann man auch einfach anstelle des 'A' ein 'LIST' angeben. Dieses 'LIST' erlaubt dann aber alle Datentypen, man könnte also auch eine Zahl ausgeben, ohne das Format zu ändern. Um Wiederholungen zu reduzieren, erlaubt PEARL für die meisten Formatierungsangaben einen Multiplikator, der in Klammern vor den mehrfach zu benutzenden Angaben steht. '(2)SKIP' steht also für 'SKIP,SKIP'. Die Ausgabe der Überschrift für die Beobachtungsergebnisse macht deutlich, daß PEARL-Anweisungen auch über Zeilengrenzen hinweggehen dürfen.

Etwas Neues finden Sie in Zeile 51, denn hier werden zum ersten Mal Zahlen ausgegeben. Die auszugebenden Werte sind I/5 und FELD(I), dem Typ nach also beides FLOAT-Zahlen (Division zweier FIXED ergibt FLOAT), die rechnerintern in Exponentialdarstellung verarbeitet werden. Durch die Angabe des Ausgabeformates 'F' werden nun die Zahlen in eine Darstellung ohne Exponenten umgewandelt, solange sie in das in Klammern angegebene Feld passen. Wollte man aber 1E-05 mit einer Formatangabe F(6) ausgeben, so würde 0.00001 nicht in das 6 Zeichen lange Feld passen und statt der Zahl '*' ausgegeben, zur Kennzeichnung, daß die Feldgröße nicht ausreichte. Der zweite Parameter gibt übrigens die Anzahl der Stellen hinter dem Komma an, die angezeigt werden.

Zeile 53 schließlich zeigt, daß man auch Zahlen und Text gemischt in einer 'PUT'-Anweisung ausgeben kann, es müssen nur die Operanden vorne und die Formatangaben hinten typmäßig übereinstimmen. Erwähnt sei hier auch das 'E'-Format, wie es in Zeile 20 auftaucht. Damit wird bestimmt, daß die zugehörigen Zahlen in Exponentialdarstellung ausgegeben werden sollen. Dabei gibt der erste Parameter in den Klammern wieder die Länge des Feldes an, in dem die Zahlen dann rechtsbündig stehen, und der zweite Parameter die Anzahl der Nachkomma-Stellen.

Zum Schluß will ich noch zeigen, wie man sich den Verwaltungskopf zunutze machen kann. Die Ausgabedaten des zweiten Beispielprogrammes machen sich zwar auf dem Bildschirm auch sehr schön, aber für diesen Artikel mußten sie ja noch irgendwie zu Papier gebracht werden. Was bot sich da eher an, als einfach die Daten statt auf den Bildschirm in ein ED-File zu schreiben. Als Ausgabestation für Daten, dargestellt im ASCII-Code, ist ein ED-File genauso geeignet, und, wie sich herausstellen sollte, wesentlich schneller. In dieser Version des Beispielprogrammes schaffte die Task REST nämlich, wie zu lesen, zwischendurch noch über 220.000 Zählrunden. Dies ist auf die schon oben erwähnte Pixelschieberei der Terminal-Emulation zurückzuführen.

Wie bewerkstelligt man eine solche Datenumleitung? Dazu war einfach nur notwendig, die Zeile 10 folgendermaßen zu modifizieren:

TERM: ED.DATA;

und schon waren die Daten vom Bildschirm verschwunden und nachträglich in dem ED-File 'ED.DATA' mit dem Editor anzusehen oder von dort auf den Drucker zu kopieren. Die ganze Änderung dauerte mit erneutem

1	1	8.41E-01
2	4	9.09E-01
3	9	1.41E-01
4	16	-7.57E-01
5	25	-9.59E-01
6	36	-2.79E-01
7	49	6.57E-01
8	64	9.89E-01
9	81	4.12E-01
10	100	-5.44E-01
11	121	-1.00E 00
12	144	-5.37E-01
13	169	4.20E-01
14	196	9.91E-01
15	225	6.50E-01
16	256	-2.88E-01
17	289	-9.61E-01
18	324	-7.51E-01
19	361	1.50E-01
20	400	9.13E-01
21	441	8.37E-01
22	484	-8.85E-03
23	529	-8.46E-01
24	576	-9.06E-01
25	625	-1.32E-01
26	676	7.63E-01
27	729	9.56E-01
28	784	2.71E-01
29	841	-6.64E-01
30	900	-9.88E-01
31	961	-4.04E-01
32	1024	5.51E-01
33	1089	1.00E 00
34	1156	5.29E-01
35	1225	-4.28E-01
36	1296	-9.92E-01
37	1369	-6.44E-01
38	1444	2.96E-01
39	1521	9.64E-01
40	1600	7.45E-01
41	1681	-1.59E-01

Rechenvariable nach jeweils
0.2 Sekunden:

0.2	1
0.4	5
0.6	9
0.8	14
1.0	18
1.2	22
1.4	26
1.6	30
1.8	34
2.0	38

Leerlaufindex: 220763

Ausgabedaten des Programms 2, umgelenkt in eine Editor-Datei.

Compiler- und Lader-Aufruf keine halbe Minute.

In dem nächsten Teil dieser Reihe geht es um verschiedene Möglichkeiten der Eingabe, weitere Mittel zur formatierten Ausgabe werden vorgestellt, das Programm greift auf das Diskettenlaufwerk zu, und die Datentypen werden genauer untersucht.

Ein kleines Adreßprogramm

Die letzte Folge dieser Reihe war dem Einstieg in die PEARL-Programmierung unter dem Betriebssystem RTOS gewidmet. Auch in dieser Folge werden die verschiedenen Sprachelemente von PEARL anhand eines Beispielprogrammes vorgestellt. Im Gegensatz zu den Demonstrationsprogrammen der letzten Folge handelt es sich aber diesmal um ein Programm mit praktischer Anwendbarkeit, nämlich um ein kleines Adreßprogramm mit den Grundfunktionen Eingabe, Löschen, sortierte Ausgabe, Suchen, Speichern auf und Lesen von Diskette. Natürlich ist es kein Ausbund an Bedienungsfreundlichkeit und Leistungsfähigkeit, doch wer mehr will, der kann ja den Grundstock beliebig erweitern, bis dann vielleicht dBase IV oder V dabei herauskommt . . .

Dieses Beispielprogramm ist umfangreicher als die bisherigen; und das stellt den PEARL-Compiler vor ein Problem. Er muß ja gleich an den Anfang des Kompilats eine Mitteilung für den Lader setzen, aus der hervorgeht, wie groß das zu ladende Modul ist. Da der UH-Compiler aber einphasig arbeitet – mit ein Grund für seine erstaunliche Schnelligkeit –, weiß er natürlich zu Beginn der Übersetzung noch nicht, wie umfangreich der Code sein wird, den er jetzt zu erzeugen hat.

Der UH-Compiler setzt deshalb 'by default' (das heißt, wenn der Benutzer es nicht anders bestimmt) eine Modulgröße von sedezimal 2000 beziehungsweise dezimal 8192 an. Bei diesem Programm läge er damit allerdings falsch; es ist größer, der Lader würde einen Speicherbereich von zu geringer Größe reservieren und anfangen zu laden, bis er den Platz belegt hat, dann aber merken, daß der Compiler ihm einen falschen Wert vorgegaukelt hat, und mit einem LOADER-INPUT-Error aussteigen.

Deshalb muß man bei größeren Programmen mit

S = hexzahl;

noch vor MODULE dem Compiler mitteilen, wie groß das Programm sein wird; die Größe ist sedezimal anzugeben. Den Wert entnimmt man der MODULE-SUMMARY, die nach jedem Compiler-Lauf ausgegeben wird. Diese enthält daneben

auch die Fehleranzahl und eine Aufzählung der Tasks und Prozeduren. Der Compiler schreibt LIMIT-ERROR in die Summary, wenn er feststellt, daß die von ihm eingesetzte oder vom Programmierer vorgegebene Größe nicht ausreicht. Man kann übrigens ein Modul auch dann laden, wenn es mit einem solchen LIMIT-ERROR übersetzt worden ist: dazu gibt es beim Lader den SZ-Parameter, mit dem die vom Compiler übergebene Größenangabe überschrieben wird. So kann man sich während der Programmentwicklung das nochmalige Kompilieren eines sonst fehlerfreien Modules ersparen.

Wenn man die in der Module-Summary angegebene Größe und die von mir vorgegebene vergleicht, so stellt man fest, daß die beiden nicht identisch sind. Aber das stört den Lader nicht. Eher noch den Benutzer, wenn er in Platznot gerät. Denn der Lader reserviert immer soviel Platz wie angegeben, egal, wieviel davon nachher tatsächlich benötigt wird. Für das (vollwertige!) Programm

```
S=E0000;
MODULE;
MODEND;
```

würde deshalb der Lader fast den gesamten Speicher des 520ST+ reservieren (896 KByte), obwohl es nur 20 Byte lang ist. Es empfiehlt sich, während der Programmentwicklung dort einen reichlich bemessenen Wert einzusetzen, um ihn nicht dauernd ändern zu müssen, der dann, wenn das Programm anscheinend fehlerfrei ist, auf den tatsächlich benötigten Wert beziehungsweise eine runde Zahl knapp oberhalb umgeändert wird.

Nun zu dem SYSTEM-Teil des Beispielprogramms: Es mag verwundern, daß vor A1 kein logischer Name für diese Datenstation steht. Das rührt daher, daß hier als logischer Name innerhalb des Modules der Name der betreffenden Datenstation selbst gewählt wurde. In diesem Fall kann man 'A1: A1;' durch 'A1;' abkürzen. Ebenso ist es der Drucker-Datenstation PP ergangen.

Zur Erläuterung der Zeile 11 muß ich einen kleinen Ausflug in das Betriebssystem machen und etwas genauer beschreiben, was bei der Eingabe von Daten über die Tastatur geschieht.

Eigentlich läuft der Vorgang ähnlich ab wie bei einer Ausgabe. Nehmen wir an, wir wären am Anfang einer Zeile. Wenn wir jetzt mit PUT irgendetwas auf den Bildschirm schreiben, so erscheint es hintereinander in einer Zeile, solange wir kein SKIP in der Formatangabe haben. Ist eine Bildschirmzeile voll, geht es in der nächsten weiter. Entsprechend läuft die Eingabe ab: es wird aus einer Eingabezeile gelesen, bis das letzte Zeichen gelesen ist, und dann geht es in der nächsten weiter. Und genau wie bei der Ausgabe kann man mit einem SKIP in die nächste Eingabezeile wechseln. So ein SKIP in einer Eingabeanweisung ist dabei nicht zu verwechseln mit einem SKIP bei der Ausgabe. Wenn in Zeile 155 vor dem A für String-Eingabe ein SKIP steht, so erfolgt die Eingabe in derselben Zeile auf dem Bildschirm, in der auch die Eingabeaufforderung steht.

Was hat man sich aber unter einer Eingabezeile vorzustellen? – Das Programm selbst kann die Tastatur nicht lesen. Die Tastatur ist für einen Prozeßrechner eine sehr langsame Datenstation mit langen Wartephasen zwischen den einzelnen Zeichen; sie wird deshalb von einer Task betreut, im Fall der Tastatur von #ACIA1.

Damit die Task, die eine Eingabe haben möchte, und die Betreuungs-Task der Eingabedatenstation miteinander in Verbindung treten können, gibt es 'Communication Elements', abgekürzt CE, die dies ermöglichen. Ein CE ist ein Speicherbereich, den die Task, die Ein- oder Ausgabe machen möchte, beim Betriebssystem anfordert. Meist bekommt sie diesen Speicher auch zugeteilt, nur wenn sie sehr viel davon anfordert, kann es vorkommen, daß dieser Wunsch nicht sofort befriedigt wird.

Wer das ausprobieren will, kann ja einmal in einer Schleife 1000mal irgendein Wort ausgeben lassen und sich unmittelbar nach dem Start mit LU den Status dieser Task ansehen. Wahrscheinlich wird dort dann 'CWS?' stehen, was soviel bedeutet wie 'waiting for communication workspace' oder aber, daß eine CE-Anforderung dieser Task nicht befriedigt wurde. Jede Task hat nämlich nur ein begrenztes Kontingent an Communication Workspace, und wenn dieses erschöpft ist, muß die Task warten, bis ein CE dadurch, daß die Betreuungs-Task es abgearbeitet hat, wieder frei wird. Für die Bildschirmausgabe des Atari bedeutet das, daß alle Zeichen auf dem Schirm erschienen sind. Dies läuft deshalb so, damit nicht eine Task, die durch einen Programmierfehler sehr viel Communication Workspace anfordert, dadurch den Rechner lahmlegt, weil der gesamte Speicher durch CE's belegt ist.

Für die Eingabe wird also von der Task ein CE angefordert und mit der Bitte, es mit Daten von der Tastatur zu füllen, an die Betreuungs-Task #ACIA1 weitergegeben. Der Datenbereich dieses CE's, der normaler-

Editieren beim Formatieren . . .?

Manchen mag gestört haben, daß der Atari ST unter RTOS-UH nicht auf Ctrl-A reagierte, wenn man eine Diskette formatierte. Dies liegt nicht etwa daran, daß der Prozessor damit vollends ausgelastet wäre, da der Atari ja zum Formatieren seine DMA-Fähigkeit einsetzen kann (im Gegensatz zu vielen anderen Rechnern), sondern daran, daß der Kommandoprozessor durch das FORM-Kommando blockiert ist.

Dies kann man umgehen, indem man das FORM-Kommando über den XC-Kanal absetzt. Dazu gibt man

COPY A1.>XC.

ein, worauf ein Gleichheitszeichen auf dem Bildschirm erscheint. Dieses Gleichheitszeichen ist der Eingabeprompt der COPY-Routine. Jetzt muß man nur noch das FORM-Kommando wie gewohnt eintippen und das zweite Gleichheitszeichen mit 'End of File' (Ctrl-D) beantworten. Danach kann man dann während des Formatierens weiterarbeiten.

weise 128 Byte lang ist, ist die Eingabezeile. Nun gibt die Betreuungstask unserer Task das CE gewöhnlich erst wieder zurück, wenn es voll ist oder aber, wenn ein 'Carriage Return', 'Line Feed' oder 'Ctrl-D' (EOT) eingegeben wurde. Soll nur ein einzelnes Zeichen von der Tastatur gelesen werden, so müßte der Nutzer also dahinter noch ein 'Return' eingeben, damit es an die Task weitergeleitet wird. Aber das wollte ich ihm nicht zumuten. Statt dessen habe ich mit (TFU=1) vereinbart, daß beim Aufruf unter dem Namen X1 ein CE an die Datenstation A1 gesandt wird, dessen Datenbereich die Länge 1 hat und damit sofort zurückkommt, wenn ein einziges Zeichen eingegeben ist. So kann man die Eingabe natürlich nicht mehr mit 'Backspace' korrigieren, und die Behandlung von Eingabefehlern ist vom Programm zu machen.

Jetzt ist auch die Wirkung des SKIP für die Eingabe etwas deutlicher: durch das SKIP wird ein neues CE angefordert. Dies ist wichtig, wenn der Nutzer einmal mehr Zeichen als gefordert eingegeben hat. Nehmen wir an, wir hätten eine Eingabe

GET STRING FROM A1 BY A(3);

und STRING sei deklariert mit CHAR(3). Gibt der Nutzer nun 'ABCJ' ein, so steht danach in STRING 'ABC'. In dem CE steht aber noch das 'J', und wenn nun, was der Nutzer ja nicht ahnen konnte, etwa die Frage

PUT 'Soll ich Ihre Arbeitsdiskette neu formatieren? (J/N) ' TO A1; GET ANTWORT FROM A1 BY A(1);

kommt, so wird dieses 'J' natürlich in ANTWORT übernommen. Weitere Erläuterungen sind damit wohl überflüssig . . .

READ und WRITE

Mit Zeile 13 öffnet sich die Verbindung zum Massenspeicher. Gerade bei einem Programm wie diesem ist man ja besonders darauf angewiesen. Durch die Zuordnung kann man jetzt über DF auf die Datei DATEI auf der Diskettenstation F0 zugreifen. Dazu kann man GET und

Bedienbefehle timer-gesteuert

RTOS/PEARL bietet die verschiedensten Möglichkeiten zur zeitlichen oder ereignisgesteuerten Einplanung von Tasks (siehe c't 6/86). Solche Einplanungen können aber nur im Zusammenhang mit den Befehlen ACTIVATE oder CONTINUE vereinbart werden. Was tun, um beliebige Befehle des Bedieninterpreters quasi timer-gesteuert einsetzen zu können?

Auch dabei machen sich der COPY-Befehl und die Datenstation XC nützlich. Mit

COPY.XXX ED.TIMER>XC.;T XXX;

veranlassen Sie, daß eine COPY-Subtask mit Namen XXX gebildet und sogleich wieder terminiert wird (noch bevor sie die zugewiesene Aufgabe in Angriff genommen hat). Die Subtask bleibt deshalb im System und kann nun beispielsweise mit

AT 15:30:30 XXX

für eine bestimmte Uhrzeit eingeplant werden. Nun brauchen Sie nur noch den Bedienbefehl, der zu diesem Zeitpunkt ausgeführt werden soll, in die Editor-Datei ED.TIMER zu schreiben. Das könnte zum Beispiel, wie im RTOS-Handbuch vorgeschlagen, eine Anweisung zum Formatieren der Floppy als netter kleiner Gruß an den zweiten Benutzer sein.

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /*   A D D R E S S   D A T A B A S E                             */
 4 /*   last update:  29.06.86     18h40                            */
 5 /*                                                               */
 6 /*****************************************************************cm*/
 7 S=ZBKK;
 8 MODULE DB;
 9 SYSTEM;
10    A1;                  /* keyboard, screen */
11    X1; A1(TFU=1);       /* for input of just 1 char */
12    PP;                  /* centronics parallel port */
13    DF; F0.DATEI;        /* address file */
14
15 PROBLEM;
16    SPC A1 DATION INOUT ALPHIC CONTROL(ALL);
17    SPC X1 DATION INOUT ALPHIC CONTROL(ALL);
18    SPC PP DATION  OUT ALPHIC CONTROL(ALL);
19    SPC DF DATION INOUT ALPHIC CONTROL(ALL);
20
21    SPC (WRITE,READ) ENTRY GLOBAL;   /* binding of global functions */
22
23    DCL TRUE   INV BIT(1) INIT ('1'B);  /* constant declarations */
24    DCL FALSE  INV BIT(1) INIT ('0'B);
25    DCL MAXADR INV FIXED  INIT (50);    /* maximal number of records */
26
27    TYPE ADDRESS STRUCT (/NAME     CHAR(20),
28                         VORNAME CHAR(20),
29                         STRASSE CHAR(20),
30                         PLZ     FIXED,
31                         ORT     CHAR(20),
32                         TELEFON CHAR(20) /);
33    DCL DATEI(50) ADDRESS;
34    DCL ORDER(50) FIXED;
35    DCL NODADR    FIXED;    /* current number of addresses in file */
36
37
38
39 /**** LTOU ***************************************************************/
40 /* ersetze klein- durch Grossbuchstaben                                */
41 /*****************************************************************/
42 LTOU:PROC (CI CHAR(1)) RETURNS(CHAR(1));
43    DCL ASC FIXED;       /* for correspondent ASCII code of character */
44
45    ASC = TOFIXED(CI);
46    IF ASC GE 97 AND ASC LE 122
47       THEN RETURN (TOCHAR(ASC - 32));
48    FIN;
49    RETURN (CI);
50 END; /* of procedure LTOU */
51
52
53 /**** LAST ***************************************************************/
54 /* suche letztes non-blank Zeichen in String                           */
55 /*****************************************************************/
56 LAST:PROC (STR CHAR(20)) RETURNS(FIXED);
57    DCL NONBL FIXED;   /* pointer in string */
58
59    NONBL = 21;
60    FOR I FROM 20 BY -1 TO 1 WHILE STR.CHAR(I) EQ ' ' REPEAT
61       NONBL = I;
62    END; /* of REPEAT loop */
63    IF NONBL EQ 1
64       THEN NONBL = 0;   /* string empty */
65    FIN;
66    RETURN (NONBL);      /* position of last non-blank + 1 */
67 END; /* of procedure LAST */
68
69
70 /**** CMPADR **************************************************************/
71 /* vergleiche Namen zweier Adressen zum Sortieren                      */
72 /*****************************************************************/
73 CMPADR:PROC (NO1 FIXED, NO2 FIXED) RETURNS (BIT(1));
74    DCL (S1,S2) CHAR(40);    /* temp. variables to reduce array access */
75    DCL (C1,C2) FIXED;       /* temp. variables for current character  */
76
77    S1 = DATEI(NO1).NAME CAT DATEI(NO1).VORNAME;
```

PUT verwenden und schreiben/lesen mit den gleichen Formatangaben wie bei der Ein-/Ausgabe über das Terminal. In diesem Programm verwende ich allerdings eine andere Zugriffsmethode, die in der jetzigen Implementation nur für die Datenstationen Floppy und Harddisk möglich ist. Es handelt sich um den Binär-Transfer von Daten mit den Funktionen WRITE und READ.

Wenn man Zahlen mit einem PUT auf die Floppy schreibt, so werden diese vor dem Schreiben in entsprechende ASCII-Codes umgewandelt, wobei die Formatangaben von Bedeutung sind. Wird etwa eine FIXED-Variable mit der Formatangabe F(8) auf die Floppy geschrieben, so werden daraus acht Bytes. Weiß man, daß eine FIXED-Variable im Speicher nur zwei Bytes belegt, möchte man natürlich auch auf der Floppy die Daten in dieser Kürze ablegen.

Dazu dienen READ und WRITE. Außerdem können dadurch Konvertierungsfehler ausgeschlossen werden, denn es muß ja beim Schreiben einmal von binär nach dezimal und beim Lesen wieder von dezimal nach binär umgerechnet werden. Ein weiterer Vorteil besteht darin, daß Felder als Ganzes geschrieben und gelesen werden können, was wesentlich schneller geht als die Übertragung elementweise.

Ein Nachteil der so erstellten Dateien ist, daß sie nicht mehr mit dem COPY-Befehl kopiert werden können. Binäre Daten dürfen ja beliebige Werte annehmen, also auch solche, die COPY als Steuerzeichen interpretiert. Wenn unter den Daten irgendwo zufällig der Wert $04 auftauchte, das ASCII-Steuerzeichen EOT (End-Of-Text), würde COPY die Übertragung

```
78    S2 = DATEI(NO2).NAME CAT DATEI(NO2).VORNAME;
79    FOR I TO 40 REPEAT
80       C1 = TOFIXED (LTOU(S1.CHAR(I)));
81       C2 = TOFIXED (LTOU(S2.CHAR(I)));
82       IF C1 LT C2
83          THEN RETURN (FALSE);  /* 0: Name 1 alphabetisch vor Name 2 */
84       FIN;
85       IF C1 GT C2
86          THEN RETURN (TRUE);  /* 1: Name 1 alphabetisch hinter Name 2 */
87       FIN;
88    END; /* of REPEAT loop */
89    RETURN (FALSE);                  /* wenn Namen gleich */
90 END; /* of procedure CMPADR */
91
92
93 /**** CONCAT ****************************************************/
94 /* fuege zwei Strings zusammen, unterdruecke abschliessende Blanks   */
95 /*****************************************************************/
96 CONCAT:PROC (STR1 CHAR(20),STR2 CHAR(20))  RETURNS(CHAR(41));
97    DCL CSTR   CHAR(41);   /* for concatenated string */
98    DCL LASTBL FIXED;      /* pointer to last non-blank char in string1 */
99    DCL LSTBL2 FIXED;      /* pointer to last non-blank char in string2 */
100
101    LASTBL = LAST(STR1);
102    LSTBL2 = LAST(STR2);
103    CSTR = STR1;
104    IF LSTBL2 NE 0            /* only if second string not empty */
105       THEN FOR I TO LSTBL2 REPEAT
106               CSTR.CHAR(I+LASTBL) = STR2.CHAR(I);
107            END; /* of REPEAT loop */
108    FIN;
109    RETURN (CSTR);            /* transfer concatenated string */
110 END; /* of procedure CONCAT */
111
112
113 /**** SORT ******************************************************/
114 /* sortiere nach Namen, speichere Reihenfolge in ORDER           */
115 /*****************************************************************/
116 SORT:PROC;
117    DCL TEMP FIXED;
118
119    FOR I TO NOADR REPEAT   /* init ORDER array */
120       ORDER(I) = I;
121    END; /* of REPEAT loop */
122    FOR I TO NOADR REPEAT
123       FOR J TO NOADR - 1 REPEAT
124          IF CMPADR(ORDER(J),ORDER(J+1))   /* swap if wrong order */
125             THEN TEMP = ORDER(J);
126                  ORDER(J) = ORDER(J+1);
127                  ORDER(J+1) = TEMP;
128          FIN;
129       END; /* of REPEAT loop */
130    END; /* of REPEAT loop */
131 END; /* of procedure SORT */
132
133
134 /**** SEARCH ******************************************************/
135 /* suche nach Datensatz mit uebereinstimmenden Namen             */
136 /*****************************************************************/
137 SEARCH:PROC (NAME CHAR(20), FIRST FIXED) RETURNS (FIXED);
138
139    FOR I FROM FIRST TO MAXADR REPEAT
140       IF NAME EQ DATEI(I).NAME
141          THEN RETURN (I);  /* matching record found */
142       FIN;
143    END; /* of REPEAT loop */
144    RETURN (0);    /* no record with matching name found */
145 END; /* of procedure SEARCH */
146
147
148 /**** GETADR ******************************************************/
149 /* hole Datei Eintrag vom Benutzer                               */
150 /*****************************************************************/
151 GETADR:PROC RETURNS(ADDRESS);
152    DCL ADR ADDRESS;
153
154    PUT 'Name:     ' TO A1 BY (2)SKIP,A;
155    GET ADR.NAME FROM A1 BY SKIP,A;
156    PUT 'Vorname: ' TO A1 BY SKIP,A;
157    GET ADR.VORNAME FROM A1 BY SKIP,A;
```

abbrechen. Ebenso würde COPY anhalten, wenn $13 (XOFF, Transmission-OFF) erscheint, und dann ewig warten auf ein erlösendes $11 (XON, Transmission-ON). Deshalb kann man mit diesem Dateiprogramm erstellte Dateien nur kopieren, indem man das Programm startet, eine Datei von der Diskette liest, die andere Diskette einlegt und dann den Befehl zum Schreiben der Datei gibt.

Die Funktionen READ und WRITE gehören nicht zu den Standardfunktionen eines PEARL-Compilers. Sie müssen deshalb als globale Funktionen spezifiziert werden. Man hat ja unter RTOS/UH die Möglichkeit, mehrere Module zusammenzubinden und Variable, Prozeduren, Funktionen und Tasks aus anderen Modulen anzusprechen. Diese müssen aber in dem Modul bekannt gemacht werden, bevor auf sie zugegriffen werden kann. Dazu dient wieder eine SPECIFY-Anweisung, die ja schon in der letzten Folge als Mittel zur Verbindung mit der Welt außerhalb eines Modules vorgestellt wurde.

Durch die Klammern zusammengefaßt, können wir WRITE und READ gemeinsam spezifizieren. ENTRY besagt, daß es sich um Prozeduren handelt, und GLOBAL, daß sie sich nicht in diesem Modul befinden. Denn man kann auch Prozeduren, Funktionen und Tasks spezifizieren, die in dem Modul sind. Dies ist notwendig, wenn ein solches Objekt hinter einer Referenz im Programmtext steht (Vorwärtsbezug). Häufig läßt sich dieses vermeiden durch Umstellung des Programmtextes, manchmal aber auch nicht, oder es ist zu mühsam, und dann wird dem Compiler eben durch ein SPECIFY mitgeteilt, daß sich hinter XYZ eine Task oder

```
158     PUT 'Strasse: ' TO A1 BY SKIP,A;
159     GET ADR.STRASSE FROM A1 BY SKIP,A;
160     PUT 'PLZ:      ' TO A1 BY SKIP,A;
161     GET ADR.PLZ     FROM A1 BY SKIP,F(6);
162     PUT 'Ort:      ' TO A1 BY SKIP,A;
163     GET ADR.ORT     FROM A1 BY SKIP,A;
164     PUT 'Telefon: ' TO A1 BY SKIP,A;
165     GET ADR.TELEFON FROM A1 BY SKIP,A;
166     RETURN (ADR);
167 END; /* of procedure GETADR */
168
169
170 /**** PUTADR ****************************************************/
171 /* gib Adresse aus                                              */
172 /*******************************************************************/
173 PUTADR:PROC (ADR ADDRESS, OUTPUT DATION OUT ALPHIC CONTROL(ALL) IDENT);
174
175     PUT CONCAT(ADR.VORNAME,ADR.NAME)
176      TO OUTPUT BY (2)SKIP,A;
177     PUT ADR.STRASSE
178      TO OUTPUT BY SKIP,A;
179     PUT ADR.PLZ,' ',ADR.ORT
180      TO OUTPUT BY SKIP,F(4),A,A;
181     PUT ADR.TELEFON
182      TO OUTPUT BY SKIP,A,SKIP;
183 END; /* of procedure PUTADR */
184
185
186 /**** LOAD ******************************************************/
187 /* hole Dateidaten von Diskette                                 */
188 /*******************************************************************/
189 LOAD:PROC;
190
191     CALL REWIND (DF);
192     CALL READ (DF,NODADR,DATEI);
193     IF NODADR GT MAXADR  /* ungueltige Groesse */
194        THEN NODADR = 0;  /* keine Adressen verfuegbar */
195             PUT 'ERROR: keine Adressdatei vorhanden'
196              TO A1 BY SKIP,A,SKIP;
197     FIN;
198 END; /* of procedure LOAD */
199
200
201 /**** SAVE ******************************************************/
202 /* speichere Datei Inhalt auf Diskette                          */
203 /*******************************************************************/
204 SAVE:PROC;
205
206     IF NODADR EQ 0
207        THEN PUT 'keine Adressen vorhanden zum Speichern'
208              TO A1 BY SKIP,A,SKIP;
209              RETURN;    /* exit procedure */
210     FIN;
211     CALL REWIND (DF);
212     CALL WRITE (DF,NODADR,DATEI);
213     CLOSE DF;
214 END; /* of procedure SAVE */
215
216
217 /**** SELECT ******************************************************/
218 /* Gib Adresse an Bildschirm aus und stelle Frage nach Kommando  */
219 /*******************************************************************/
220 SELECT:PROC (ADR ADDRESS,FRAGE CHAR(32),EXIT BIT(1) IDENT)
221              RETURNS(CHAR(1));
222     DCL CHR CHAR(1);
223
224     CALL PUTADR (ADR,A1);
225     PUT FRAGE TO A1 BY SKIP,A;
226     GET CHR FROM A1 BY SKIP,A;
227     CHR = LTOU(CHR);
228     IF CHR EQ 'E'
229        THEN EXIT = TRUE;
230     FIN;
231     IF CHR EQ 'D'
232        THEN CALL PUTADR(ADR,PP);
233     FIN;
234     RETURN (CHR);
235 END; /* of procedure SELECT */
236
237
```

Prozedur verbirgt, die später noch auftaucht. Kommt sie nicht vor, ist dies natürlich ein Fehler.

Der Datentyp BIT

Was in Pascal 'boolean', ist in PEARL BIT(1), denn natürlich braucht man auch hier die Möglichkeit, logische Ergebnisse zu speichern. Die Deklaration einer solchen BIT-Variablen ist in Zeile 243 zu sehen. Aus der Klammer mit der Zahlenangabe 1 läßt sich ersehen, daß man auch BIT-Variablen größerer Länge deklarieren kann, BIT(8) enthielte 8 Bit, die dann aber gemeinsam keine logische Bedeutung mehr im Sinne von 'wahr' und 'falsch' haben. 'Wahr' ist eine BIT(1)-Variable, wenn sie den Wert '1'B enthält und 'falsch', wenn sie '0'B enthält. Allgemein werden Binärwerte durch Hochkommata eingeschlossen und durch ein angehängtes B gekennzeichnet. Zur Vereinfachung kann man Binärwerte aber auch oktal mit angehängtem B3 oder sedezimal mit angehängten B4 schreiben. Folgende Werte sind also gleich: '01011110'B, '136'B3, '5E'B4.

Zur Verbesserung der Lesbarkeit habe ich 'wahr' und 'falsch' in den Zeilen 23 und 24 als Konstanten TRUE und FALSE deklariert. Dazu wird in die normale Variablendeklaration ein INV (invariant) hinter dem Namen eingefügt und eine Anfangswertangabe INIT(anfangswert) angehängt. Mit einer solchen Anfangswertangabe kann man auch Variablen initialisieren (Zeile 242).

MAXADR ist schließlich eine Konstante, die angibt, wie viele Adressen man mit diesem Programm verarbeiten kann. Um diese Anzahl zu verdoppeln, wäre hier und in den Felddeklarationen Zeile 33 und 34 100

```
238 /**** DB ****************************************************************/
239 /* Adress Verwaltungs Programm                                          */
240 /******************************************************************/
241 DB:TASK;
242    DCL DONE BIT(1) INIT('0'B);
243    DCL EXIT BIT(1);
244    DCL INDEX FIXED;
245    DCL NAME CHAR(20);
246    DCL FRAGE CHAR(32);
247    DCL CHR  CHAR(1);
248
249    NODADR = 0;        /* noch keine Adressen verfuegbar */
250    WHILE NOT DONE REPEAT    /* Kommando Schleife */
251       /* Gib Menue aus */
252       PUT 'Adressen Verwaltung' TO A1 BY (2)SKIP,A,SKIP;
253       PUT 'A: Adressen zeigen' TO A1 BY SKIP,A;
254       PUT 'B: Adresse suchen' TO A1 BY SKIP,A;
255       PUT 'C: Adresse eingeben' TO A1 BY SKIP,A;
256       PUT 'D: Adresse loeschen' TO A1 BY SKIP,A;
257       PUT 'E: Adressdatei lesen' TO A1 BY SKIP,A;
258       PUT 'F: Adressdatei speichern' TO A1 BY SKIP,A;
259       PUT 'G: Ausgang' TO A1 BY (2)SKIP,A;
260       PUT 'Wahl:  ' TO A1 BY (2)SKIP,A;
261       /* Hole Kommando */
262       GET CHR FROM X1 BY SKIP,A;
263       /* Fuehre Kommando aus */
264       EXIT = FALSE;
265       CASE TOFIXED(LTOU(CHR)) - TOFIXED('A') + 1
266
267          ALT /* A */   /* Adressen zeigen */
268             CALL SORT;
269             FRAGE = 'Weiter (''J''a,''E''xit,''D''rucken): ';
270             FOR I TO NODADR WHILE NOT EXIT REPEAT
271                CHR = SELECT (DATEI(ORDER(I)),FRAGE,EXIT);
272             END;
273
274          ALT /* B */   /* Adressen suchen */
275             PUT 'Name: ' TO A1 BY (2)SKIP,A;
276             GET NAME FROM A1 BY SKIP,A;
277             FRAGE = 'Weiter (''J''a,''E''xit,''D''rucken): ';
278             INDEX = 1;
279             WHILE NOT EXIT REPEAT
280                INDEX = SEARCH(NAME,INDEX);
281                IF INDEX EQ 0
282                   THEN EXIT = TRUE;
283                   ELSE CHR = SELECT (DATEI(INDEX),FRAGE,EXIT);
284             FIN;
285             INDEX = INDEX + 1;
286          END; /* of WHILE loop */
287
288          ALT /* C */   /* Adresse eingeben */
289             NODADR = NODADR + 1;
290             IF NODADR GT MAXADR
291                THEN PUT 'Datei voll' TO A1 BY SKIP,A,SKIP;
292                     NODADR = NODADR - 1;
293                ELSE DATEI(NODADR) = GETADR;
294             FIN;
295
296          ALT /* D */   /* Adresse loeschen */
297             FRAGE = 'Loeschen (''J''a,''N''ein,''E''xit): ';
298             FOR I TO NODADR WHILE NOT EXIT REPEAT
299                IF 'J' EQ SELECT (DATEI(I),FRAGE,EXIT)
300                   THEN DATEI(I) = DATEI(NODADR);
301                        NODADR = NODADR - 1;
302                FIN;
303             END;
304
305          ALT /* E */   /* Adressdatei von Diskette lesen */
306             CALL LOAD;
307
308          ALT /* F */   /* Adressdatei auf Diskette schreiben */
309             CALL SAVE;
310
311          ALT /* G */   /* Programm beenden */
312             DONE = TRUE;
313             PUT 'Bye' TO A1 BY (2)SKIP,A,SKIP;
314
```

einzusetzen. Es sei hier darauf hingewiesen, daß es sehr zur Übersichtlichkeit eines Programms beiträgt, wenn man wichtige Konstanten zentral mit einem Label vereinbart. So kann man sie bei Bedarf schnell ändern, ohne die Gefahr, bei mehrfachem Auftreten im Programmtext etwas zu übersehen.

CHAR-Variable

Neben dem Rechnen ist heute die Textverarbeitung ein bedeutender Anwendungsbereich für Computer. Auch hierfür stellt PEARL leistungsfähige Sprachelemente zur Verfügung. Doch zunächst zur Speicherung:

Texte werden in CHARACTER- oder kürzer CHAR-Variablen gespeichert, denen in Klammern eine Längenangabe folgen kann, ähnlich der BIT-Variablen oder der Feldbeschreibung eindimensionaler Felder. Eine CHAR-Variablen-Deklaration ist in Zeile 74 zu sehen. Dort werden zwei Strings von je 40 Zeichen Länge deklariert. Die Regeln für das Kopieren verschieden langer Strings sind recht einfach: was nicht paßt, wird rechts abgeschnitten beziehungsweise durch Leerzeichen aufgefüllt. Kopiert man also 'ABCDEF' in eine CHAR(4)-Variable, so enthält sie 'ABCD', kopiert man 'GH' in diese, so enthält sie 'GH '. Auf diese Weise ist CSTR in Zeile 103 trotz Zuweisung eines nur 20 Zeichen langen Strings in ganzer Länge mit definiertem Inhalt versehen.

Will man zwei Strings aneinanderhängen, gibt es dazu die Funktion CAT; Beispiele sind in den Zeilen 77 und 78 zu sehen.

Man kann aber nicht nur Strings als Ganzes bearbeiten, sondern über den Selektor .CHAR(integerexpression)

```
315            OUT /* invalid command */
316                : /* do nothing */
317
318         FIN; /* of CASE */
319     END; /* of REPEAT loop */
320 END; /* of task DB */
321
322 /*****************************************************************/
323 MODEND;
```

Zum Kennenlernen von PEARL: ein komplettes kleines Dateiverwaltungsprogramm

auch jedes einzelne Zeichen aus einem String herausfischen. Ein schönes Beispiel, auch für einen Wiederholungsblock, ist die Funktion LAST in Zeile 53ff. Diese Funktion sucht nach der Position in einem String, ab der nur noch Leerzeichen darin stehen. Schön ist hier die Verknüpfung von Zählschleife und Abbruchbedingung zu sehen, wobei die Zählvariable in der Bestimmung der Abbruchbedingung auftaucht. Die Position wird gefunden, indem die Zeichen des Strings von hinten beginnend mit dem Leerzeichen verglichen werden. Taucht das erste Zeichen ungleich dem Leerzeichen auf, enthält NONBL aus der Runde vorher die gewünschte Position.

Wie in Zeile 231 und 140 zu sehen, kann man Strings mit dem Operator EQ (equal) direkt auf Gleichheit untersuchen. Etwas anders sieht es aus, wenn man die Reihenfolge von Buchstaben gemäß dem Alphabet untersuchen will. Die Operatoren GT (greater than), GE (greater or equal), LT (less than) und LE (less or equal) sind nämlich für Buchstaben nicht zugelassen. Das liegt daran, daß PEARL-Programme nicht grundsätzlich für Computer geschrieben werden, die intern mit einer ASCII-Darstellung der alphanumerischen Zeichen arbeiten. Und so ein Vergleich würde etwa im EBCDIC-Code ein gänzlich anderes Ergebnis haben. Deshalb sind hierfür die Buchstaben mit TOFIXED in die entsprechenden ASCII-Codes, also Zahlen, zu konvertieren und diese dann zu vergleichen. Die Umkehrung von TOFIXED ist TOCHAR (Zeile 47), das aus einem Integerwert einen Buchstaben gemäß ASCII macht.

Verbund-Variablen

Haben mehrere Variablen einen gemeinsamen Bezug, so faßt man sie günstig in Verbund-Variablen zusammen. Taucht eine Verbund-Variable mehr als einmal auf, ist es sinnvoll, einen entsprechenden Typ zu definieren. Verbunde oder Strukturen, in Pascal 'records', sind nichts anderes als die Zusammenfassung von Variablen, auch verschiedenen Typs, unter einem gemeinsamen Namen. Zu einer Adresse zum Beispiel gehören verschiedene Elemente, die nur gemeinsam den sinnvollen Inhalt 'Adresse' haben. Deshalb ist in Zeile 27 bis 32 der neue Datentyp ADDRESS definiert, der die aufgezählten Elemente enthält.

Dieser neue Datentyp wird in Zeile 33 gleich zur Deklaration eines Feldes gebraucht, das 50 Adressen umfaßt. Dieses Feld enthält die Datensätze, mit denen in diesem Programm gearbeitet wird. Will man auf ein einzelnes Element eines Verbundes zugreifen, so ist an den Namen der Verbund-Variablen mit einem '.' der Elementname anzuhängen. So gestattet DATEI(17).TELEFON den Zugriff auf den String TELEFON im 17. Datensatz von DATEI. Auch das Schachteln von Verbunden ist gestattet. So dürfte, wenn notwendig, in der Adresse auch eine Variable vom Typ KOMPLEX folgendermaßen definiert werden:

TYPE KOMPLEX (/RE FLOAT, IM FLOAT/);

Man kann Verbunde als Ganzes zuweisen (Zeile 300). Dazu muß aber eine Typ-Definition gemacht und die Verbund-Variable mit diesem Typ deklariert worden sein. Dies ist auch notwendig, wenn Verbunde an Funktionen und Prozeduren übergeben werden sollen oder eine Funktion einen Verbund als Resultat liefert (Zeile 151).

Prozeduren und Funktionen

Funktionen und Prozeduren beginnen mit ihrem Namen. Es folgt hinter einem Doppelpunkt das Schlüsselwort PROC, dann eine Parameterliste, die leer sein darf (Zeile 189), und bei Funktionen muß schließlich noch der Typ des Resultates angegeben werden. Prozeduren werden mit CALL XYZ aufgerufen. Funktionsaufrufe dürfen überall dort stehen, wo auch eine Variable gleichen Typs stehen dürfte (Zeile 271, 299). Die Typangabe bei Funktionen erfolgt mit RETURNS(typangabe) (Zeile 42).

Dabei darf der Typ auch ein Verbund sein, wie man in Zeile 151 sieht. Diese Funktion liest eine Adresse ein und übergibt sie dann komplett der aufrufenden Task (Zeile 293).

Hinter Parameterliste und Typangabe taucht in manchen PEARL-Lehrbüchern noch die Angabe REENT auf. Dies steht für 'reentrant' ('wiedereintrittsfähig') und bedeutet, daß die Funktion oder Prozedur von mehreren Tasks gleichzeitig aufgerufen werden darf. Unter RTOS/UH-PEARL ist diese Angabe überflüssig, da hier Funktionen und Prozeduren immer 'reentrant' sind. Diese Eigenschaft wird dadurch erreicht, daß Variable nur indirekt angesprochen werden und dadurch mit jedem Aufruf ein neuer Speicherbereich für die Variablen angelegt werden kann. So einen Prozedur- oder Funktions-Variablenbereich erkennt man an dem Kürzel PWSP, wenn man sich den Systemstatus mit dem 'S'-Kommando anzeigen läßt.

In der Parameterliste stehen die Parameter, die die aufrufende Task, Prozedur oder Funktion bekanntmacht, ansonsten sind nur globale Variable innerhalb einer Prozedur oder Funktion bekannt. Die Parameter werden übergeben durch Angabe eines Namens für den Bereich innerhalb der Prozedur oder Funktion und die Datentypangabe (Zeile 73). Mehrere Parameter werden durch Kommata getrennt. Werden auf diese Weise Parameter übergeben, so geschieht eine Übergabe 'by value', das heißt, daß eine eigene Speicherzelle für diese Variablen im Procedure Workspace (PWSP, s.o.) bereitgehalten wird, in die beim Aufruf der Variableninhalt des Aufrufers kopiert wird.

Auf diese Weise kann eine Prozedur oder Funktion keine Variablen außerhalb verändern, weder gewollt noch ungewollt. Manchmal möchte man aber gerade dieses; dann kann man Parameter 'by identity' übergeben, was durch den Zusatz IDENT in der Parameterliste geschieht (Zeile 220). In diesem Fall wird die Adresse der zugehörigen Variablen des Aufrufers übergeben; die Prozedur/Funktion arbeitet dann mit dieser Variablen, als sei es eine eigene. So etwas ist nützlich, wenn mehr als eine Variable Ergebnis einer Funktion sein soll oder eine Prozedur andere als globale Variable bearbeiten soll. Die Funktion SELECT etwa übergibt einen Buchstaben als Funktionswert mit RETURN(funktionswert) (Zeile 234), während das Ergebnis der logischen Verknüpfung 'Buchstabe ist gleich 'E'' dadurch dem Aufrufer bekannt ist, daß er der Funktion eine BIT(1)-Variable 'by identity' übergibt.

Eine weitere Besonderheit besteht darin, daß man auch Datenstationen als Parameter übergeben darf. Diese müssen 'by identity' übergeben werden (Zeile 173). Auf diese Weise ist nur eine einzige Prozedur, nämlich PUTADR, nötig, um die Ausgabe sowohl auf den Bildschirm als auch auf den Drucker zu besorgen. Es müssen nur die übergebenen Datenstationen mit der Spezifikation in der Parameterliste übereinstimmen, wobei es nicht stört, daß für A1 auch die Datenrichtung Eingabe zugelassen ist, während in der Parameterliste nur die Datenrichtung Ausgabe gefordert wird.

An dieser Stelle sei auf eine Abweichung des RTOS/UH-PEARL-Compilers vom DIN-Standard hingewiesen. Nach DIN ist es möglich, in Tasks, Prozeduren und Funktionen lokale Variablen zu deklarieren mit Namen, die auch globale Variablen haben. Eine global deklarierte ZAHL FIXED wäre damit verschieden von einer lokal deklarierten ZAHL FIXED. Da es verwirren kann, wenn in zwei Tasks dann ZAHL auftaucht, zwei vollkommen verschiedene Variable aber damit gemeint sind, ist dies bei dem RTOS/UH-PEARL-Compiler nicht möglich. Wird es dennoch versucht, so bekommt man einen /DOUBLE-DEF/initION-Fehler. Natürlich dürfen lokale Variablen in verschiedenen Tasks, Prozeduren und Funktionen gleiche Namen haben, die Einschränkung gilt nur für die Namen globaler Variablen.

CASE-Auswahl

Das Beispiel-Programm arbeitet menügesteuert. Dies bedeutet, daß das Drücken einer einzigen Taste genügt, um eine Funktion des Programmes aufzurufen. Um nun den Aufruf der verschiedenen Funktionen einfach zu gestalten, wurde jeder Funktion in aufsteigender Folge ein Buchstabe zugeteilt. Dadurch ist die Auswahl mit CASE einfach. Auf CASE muß ein Integerwert folgen, der angibt, welches ALT angesprungen wird. Ist der Integerwert 1, so wird das erste ALT angesprungen, ist er 2, so wird das zweite ALT angesprungen und so weiter. Deshalb muß noch eine Anpassung erfolgen, da 'A' ja durch die Funktion TOFIXED den Wert 65 liefert. Dies geht relativ einfach auch ohne Wissen um die absoluten ASCII-Codes: Der ASCII-Code des eingegebenen Zeichens wird zunächst auf Kleinbuchstaben getestet (natürlich frißt dieses Programm auch Kleinbuchstaben als Funktionsaufruf) und

eventuell zum Großbuchstaben konvertiert; dann wird der ASCII-Code von 'A' abgezogen und 1 addiert.

Zahlen, die größer als die Anzahl der ALT's sind, führen ebenso wie 0 dazu, daß die Routine hinter OUT angesprungen wird; diese sollte deshalb die Fehlerbehandlung enthalten, wenn diese Werte nicht auftreten dürfen. Hier treten solche Werte dann auf, wenn der Nutzer andere als die angegebenen Tasten drückt; das verzeiht das Programm ohne Bemerkung, es wird einfach das Menü noch einmal ausgegeben.

Nicht zu vergessen ist ein FIN als Abschluß des ganzen Gebildes, ähnlich IF .. THEN .. ELSE .. FIN.

Maus, Grafik, Interrupts

Atari ST ohne Maus? Wem diese lieb ist, der soll sie auch unter RTOS nicht missen. Deshalb stelle ich jetzt ein kleines Zeichenprogramm vor, das die Maus als Stift benutzt. Von den Leistungsmerkmalen her ist es vergleichbar dem Programm 'Doodle' unter TOS. Man kann damit zeichnen, radieren, Bilder löschen, auf Diskette abspeichern und von dieser wieder laden. Nur der Aufruf dieser Funktionen geschieht auch weiterhin über die Tastatur.

Nach der Tastatur ist die Maus die zweite Quelle, von der Signale aus der Außenwelt an den Rechner gelangen. Dabei besteht, zumindest beim ST, intern gar kein so großer Unterschied zwischen Tastatur und Maus. Beide Arten von Signalen werden von dem unter der Tastatur sitzenden Prozessor 6301 'vorverdaut' und dem Hauptprozessor über eine serielle Schnittstelle in mundgerechten Happen gereicht.

Eine besondere Eigenschaft dieser seriellen Schnittstelle ist ihre Fähigkeit, einen Interrupt (Programmunterbrechung) auszulösen. Diese ermöglicht es, daß der Hauptprozessor kontinuierlich arbeiten kann und nicht immer wieder nachschauen muß, ob sich in seinem 'Briefkasten' schon wieder ein Kommando des Benutzers befindet: Erst wenn eine Taste gedrückt oder die Maus bewegt wird, macht ein Interrupt den Hauptprozessor darauf aufmerksam, daß es wieder Informationen von der seriellen Schnittstelle abzuholen gibt.

Die Maus – ein externes Ereignis

Der 6301 hat 4 KByte ROM, in das das Programm zur 'Vorverdauung' von Tastatur, Maus und eventuell Joystick eingebrannt ist. Deshalb werden die von dort stammenden Informationen immer in einem der programmierten Formate angeboten, egal, ob der Rechner nun unter TOS oder unter RTOS betrieben wird. Unter RTOS werden aber die Informationen über Mausbewegungen normalerweise nur in einen Puffer geschrieben und nicht weiterverarbeitet. Für die bedarfsgerechte Weiterverarbeitung muß deshalb erkannt werden können, daß Daten über eine Mausbewegung in dem Puffer abgelegt wurden, und dieser Puffer muß zugänglich sein.

PEARL als Sprache der Prozeßrechner besitzt Sprachmittel, um auf Interrupts in unterschiedlicher Weise zu reagieren. Zunächst muß aber eine Interrupt-Quelle bekannt gemacht werden. Dies geschieht unter RTOS in dem Systemteil durch Angabe eines Namens, mit dem der Interrupt dann angesprochen werden kann, gefolgt von einem Doppelpunkt und 'EV(hexnum)'. 'hexnum' ist ein 32stelliges Bitmuster, bei dem sich hinter jedem Bit eine Interrupt-Quelle verbergen kann. Mit 'EV(80000000)' (Zeile 16) wird eine 'Maske' gesetzt, die den Interrupt hinter Bit 31 (die Numerierung beginnt rechts mit Bit 0) freigibt, den Maus-Interrupt. Auch Bit 30 ist bei der ST-Implementation belegt, allerdings nicht durch ein externes Ereignis, sondern durch den rechnerinternen Bild-Synchronisationsimpuls für den Monitor. Im Problemteil ist ein Interrupt zu spezifizieren mit dem Schlüsselwort 'INTERRUPT' (Zeile 24).

Das Bekanntmachen eines Interrupts reicht aber noch nicht aus. Er muß noch freigegeben werden, und außerdem will das Betriebssystem wissen, was passieren soll, wenn der Interrupt auftritt. Die Freigabe geschieht durch 'ENABLE interruptname' (Zeile 177), das Sperren entsprechend durch 'DISABLE interruptname' (Zeile 83). Der

Benutzer kann Interrupt-Quellen auch von der Kommandoebene des Betriebssystems aus freigeben und sperren; allerdings muß er dann natürlich anstelle des programminternen Interrupt-Namens das Schlüsselwort 'EV' und die zugehörige Bitmaske angeben. 'EV' steht übrigens für 'event', was soviel wie Ereignis bedeutet.

Doch nun zur Hauptsache. Durch den Interrupt soll ja eine Aktion des Rechners ausgelöst werden. PEARL bietet dazu eine sprachliche Konstruktion in der Form 'WHEN interrupt-name ACTIVATE ...' an, die dazu dient, eine Task als Reaktion auf einen Interrupt einzuplanen (Zeile 179). Die Wirkung ist ebenso wie bei der zyklischen Einplanung mit 'ALL 0.05 SEC ACTIVATE...', an die Sie sich vielleicht aus dem Teil 3 dieser Serie erinnern: Jedesmal, wenn das auslösende Ereignis eintritt, wird die eingeplante Task durch das Betriebssystem gestartet – nur daß es sich in diesem Fall nicht um den periodischen Ablauf einer Zeitspanne, sondern um den Maus-Interrupt handelt, der bei jeder Bewegung ausgelöst wird. Deshalb ist dem Mäusegeist, wenn einmal gerufen, auch nur mit 'PREVENT MOUSE' oder aber ganz radikal durch 'UNLOAD MOUSE' beizukommen.

Sperrt man nur mit 'DISABLE EV 80000000' den Interrupt, bleibt die Einplanung erhalten. Dies nutze ich in der Task MOUSE aus. Diese Task braucht eine gewisse Zeit zur Verarbeitung einer Mausbewegung beziehungsweise eines Tastendruckes. Während dieser Zeit würde der kleine Nebenprozessor schon weitere Daten liefern, die jeweils wieder einen Interrupt verursachen und RTOS veranlassen würden, eine Aktivierung der Task MOUSE vorzumerken. Ab der vierten Vormerkung wird aber ein 'INTERRUPT OVERFLOW' ausgegeben, der das System stark blockiert; denn die #ERROR-Task, die eine solche Meldung fabriziert, hat eine hohe Priorität, und die Bildschirmausgabe über die Terminal-Emulation kostet beim ST viel Zeit, was zu einer Eskalation führen kann, da die Task MOUSE noch weniger zum Zuge kommt und noch mehr Aktivierungen 'überhängig' werden. Deshalb wird zu Beginn der Task MOUSE der Maus-Interrupt gesperrt; die Mausbewegung wird in Ruhe verarbeitet, und gegen Ende der Task wird der Interrupt dann wieder freigegeben.

Mehrere Bilder gleichzeitig

Nach diesen einleitenden Bemerkungen zum Maus-Interrupt jetzt zum Programm. Voraussetzung für dieses Programm sind die bereits in die EPROMs eingebrannten Grafikfunktionen. Diese Funktionen gehören alle nicht zum Standard-Sprachumfang von PEARL, stehen aber bei PEARL-UH zur Verfügung (zum Teil nur für den Atari ST). Sie teilen sich auf in Einbaufunktionen, die dem Compiler bekannt sind (SETPIX und LINE), und erweiterte Einbaufunktionen, die vor ihrer Benutzung als externe Prozeduren spezifiziert werden müssen (Zeile 25,26). Ihre Funktion werde ich jeweils beim erstmaligen Auftreten erklären.

Den Einstieg in das Zeichenprogramm bildet die Task START. Sie beginnt mit dem Aufruf von GETNAM, einer Prozedur, die der String-Variablen MNAME den Namen 'Bild ' oder einen vom Nutzer einzugebenden Namen zuweist. Dabei wird die Funktion OPSTR benutzt, die es ermöglicht, mit 'CR' einen vorgegebenen String zu bestätigen (dann enthält STRING nach Zeile 52 nur Leerzeichen) oder zu ersetzen.

Der Bildname wird für die Einbaufunktion SCRSET gebraucht. Mit dieser Funktion wird ein Speichermodul eingerichtet beziehungsweise der Malzeiger der aufrufenden Task auf dieses gesetzt, wenn es schon existierte. Der Atari ST verwendet ja als Videospeicher den Hauptspeicher, in dem der Videopointer auf einen 32 KByte großen Speicherbereich zeigt, dessen Speicherinhalt als Pixelmuster dann auf dem Bildschirm zu sehen ist. Mit SCRSET wird so ein Speicherbereich angefordert, der dann mit '#name ' in der Speicherverwaltung erscheint. Das Doppelkreuz bedeutet dabei, daß dieses Modul nicht mit 'UNLOAD' wieder entfernt werden kann. Dazu ist die Funktion SCKILL zu benutzen, die auch von der Kommandoebene direkt aufgerufen werden kann. Der Name eines Bildmoduls darf aus maximal fünf Buchstaben bestehen.

Mit SCRSET können soviele Bildschirmseiten eingerichtet werden, wie im Speicher Platz ist, beim 'Mega-Atari' also bis zu 30. Aber nicht nur zum Einrichten der Bildschirme wird SCRSET gebraucht: Wie schon angedeutet, wird mit dieser Funktion auch ein Malzeiger gesetzt. Dieser gibt für die Funktionen SETPIX, LINE und CLEAR an, auf welchem Bildschirm gearbeitet werden soll, um den Verwaltungsaufwand zu verringern und die Grafik hinlänglich schnell zu machen. Es gibt für jede Task einen Malzeiger, der natürlich jeweils nur auf einen Bildschirm gleichzeitig zeigen kann. Soll auf einen anderen Bildschirm gewechselt werden, ist deshalb

Mausanschluß? - Unter RTOS-UH/PEARL kein Problem

wiederum SCRSET zu bemühen. Vor dem Aufruf von SCRSET zeigt der Malzeiger auf den Hauptbildschirm, der mit dem Namen '#MAIN ' angesprochen werden kann.

Der frisch eingerichtete Bildschirm wird mit der Funktion CLEAR (Zeile 167) gelöscht. Es wird aber immer noch der Hauptbildschirm angezeigt. Mit SCRCHG, auch einer Funktion, die von der Kommandoebene aufgerufen werden kann, wird nun der Videopointer auf den neuen Bildschirm 'umgebogen', und es ist dieser strahlendleer zu sehen. Würde an dieser Stelle das Programm angehalten, hätte man blind den Befehl 'ctrl-A SCRCHG #MAIN' einzugeben, um zu dem Hauptbildschirm zurückzugelangen. Da dies aber zu ungeahnten Folgen führen kann, weil etwa gleichzeitig der Editor auf dem Hauptbildschirm läuft, ist ein bequemes Zurückschalten zum Hauptbildschirm auch mit der Taste 'HELP' möglich. Mit 'Shift-HELP' geht es dann wieder zurück zum zuletzt aktiven Bildschirm.

Da RTOS auch Farbgrafik mit mittlerer Auflösung unterstützt, kann man mit der Funktion WIDTH die Anzahl der Pixel in x- und y-Richtung abfragen. Nach dem Funktionsaufruf finden sich in den beiden FIXED-Variablen zum Beispiel für meinen monochromen Monitor die Werte 640 und 400. Um den Pfeil, der die Maus auf dem Bildschirm repräsentieren soll, nicht aus dem Bild wandern zu lassen, werden XLIMIT und YLIMIT auf eine Pfeillänge weniger gesetzt. MX und MY enthalten die aktuelle Mausposition, die in der Bildschirmmitte aufgesetzt wird (Zeile 172,173), und MXO und MYO die vorangehende Mausposition.

Das Bild soll nicht zerstört werden, wenn sich der Pfeil darüber hinwegbewegt; deshalb muß der betroffene Bildschirmausschnitt vorher abgespeichert und hinterher wieder restauriert werden. Dazu dient das Funktionenpaar SPRITL und SPRITS. Mit diesen Funktionen kann ein Bildschirmausschnitt in einem zweidimensionalen Bit-Feld gespeichert und auch wieder von dort zurückgeholt werden. Lage und Größe des Ausschnitts werden beschrieben, indem man die Koordinaten der linken oberen Ecke und, darauf bezogen, die Ausdehnung in x- und y-Richtung angibt. Beachtung verdient dabei die Dimension des Bit-Feldes in x-Richtung, da wortweise (BIT(16)) gespeichert wird. Sie muß immer ein Wort mehr betragen, als zur Speicherung der Bit-Anzahl in x-Richtung notwendig wäre, da ja nicht unbedingt der Bildschirmausschnitt mit einer Wortgrenze im Speicher zusammenfallen muß. Ist zum Beispiel ein Bildschirmausschnitt in x-Richtung 67 Bit lang, so wären zu dessen Speicherung fünf Worte notwendig, und das Bit-Feld müßte in x-Richtung die Dimension 6 haben. Bei Farbgrafik ist dieser Wert noch mit 2 zu multiplizieren.

In der Task START werden außerdem der Maus-Interrupt freigegeben, die Task SCRUP gestartet und die Task MOUSE für das Auftreten des Maus-Interruptes eingeplant. Die Task SCRUP ist jetzt also lauffähig, und die Task MOUSE wartet auf ihren Interrupt. Auf diesen Interrupt will ich sie aber noch ein wenig warten lassen und erst die Task SCRUP beschreiben.

Synchronisiert zeichnen

SCRUP ist die Task, die die Mausbewegungen auf dem Bildschirm darstellt und das Zeichnen und Löschen vornimmt. Löschen ist dabei nichts anderes als zeichnen mit der Hintergrundfarbe. Dazu muß der Malzeiger auf den Zeichenbildschirm gerichtet sein, was in Zeile 130 durch SCRSET geschieht. Alles folgende läuft in

einer Endlosschleife ab, wie sie wohl noch aus Teil 3 dieser Serie erinnerlich ist. An dieser Stelle enthalten MXO und MYO die Koordinaten, auf denen die Maus (als Pfeil) gerade auf dem Bildschirm dargestellt wird, und MX und MY die Koordinaten, zu denen sie bewegt wurde; der Bildschirmausschnitt unter der Maus ist in ARROWS gespeichert, und LBTN und RBTN sind 'wahr', wenn die linke beziehungsweise rechte Maustaste gedrückt ist.

Aus noch nicht ersichtlichen Gründen wird zunächst der Interrupt, der durch den Vertikal-Bildsynchronisationsimpuls ausgelöst wird, gesperrt (Zeile 132). Doch dazu später. Mit der schon beschriebenen Funktion SPRITS wird der Bildschirmausschnitt unter dem Pfeil restauriert, es ist also jetzt kein Pfeil mehr zu sehen. Abhängig davon, ob eine der Maustasten gedrückt ist, wird jetzt eventuell eine Linie von der alten zur neuen Mausposition gezogen. Ist die rechte Maustaste gedrückt, so wird keine Linie gezogen und auch nicht gelöscht, die Maus läßt sich also frei bewegen. Außerdem wird die linke Taste ignoriert. Ist die rechte Taste aber nicht gedrückt, so wird eine Linie gezogen, wenn die linke Taste nicht gedrückt ist (Zeile 139), beziehungsweise gelöscht, wenn sie gedrückt ist. LINE wird dazu der Anfangs- und der Endpunkt der Linie übergeben sowie die Farbe, mit der sie gezogen werden soll.

Da das Zeichnen und Löschen mit nur einem Pixel Breite geschieht, ist es ziemlich schwierig, Flächen geschlossen schwarz oder weiß zu bekommen. Abhilfe läßt sich recht einfach schaffen, indem man mehrere Linien parallel zeichnen läßt, jeweils um ein Pixel nach rechts oder unten verschoben. Die fol-

```
=   1 /!!!
```

```
61
62
63 /!!!! GETNAM !!!!
```

```
= 123 END;  /* of task MOUSE */
= 124
= 125
= 126 /*** SCRUP ****************************************************************/
= 127 /*  Screen aktualisieren                                                  */
= 128 /*****************************************************************************/
= 129 SCRUP:TASK PRIO 16 RESIDENT;
= 130    CALL SCRSET (MNAME);
= 131    REPEAT                    /* endless loop */
= 132       DISABLE SYNC;
= 133       CALL SPRITS(MIO,MYO,10,10,MNAME,ARROWS);
= 134       REQUEST SCRSEM;
= 135       IF NOT RBTN            /* right button: don't draw nor erase - */
= 136          THEN
= 137             IF LBTN         /* if left button pressed   */
= 138                THEN CALL LINE (MIO,MYO,MI,MY,0);   /* then erase   */
= 139                ELSE CALL LINE (MIO,MYO,MI,MY,1);   /* else draw   */
= 140             FIN;
= 141       FIN;
= 142       MIO = MI;              /* save mouse position as old   */
= 143       MYO = MY;
= 144       CALL SPRITL(MI,MY,10,10,MNAME,ARROWS);
= 145       /* draw arrow   */
= 146       CALL LINE (MI + 1,MY + 1,MI + 8,MY + 8,1);
= 147       CALL LINE (MI + 5,MY + 4,MI + 9,MY + 8,0);
= 148       CALL LINE (MI + 4,MY + 5,MI + 8,MY + 9,0);
= 149       CALL LINE (MI + 2,MY + 2,MI + 4,MY + 6,1);
= 150       CALL LINE (MI   ,MY   ,MI + 3,MY + 6,0);
= 151       CALL LINE (MI + 2,MY + 2,MI + 6,MY + 4,1);
= 152       CALL LINE (MI   ,MY   ,MI + 6,MY + 3,0);
= 153       CALL SETPIX(MI + 9,MY + 9,0);
= 154       RELEASE SCRSEM;
= 155       WHEN SYNC RESUME;
= 156    END;  /* of endless loop */
= 157 END;  /* of task SCRUP */
= 158
= 159
= 160 /*** START ****************************************************************/
= 161 /*  Einrichten der Maus                                                   */
= 162 /*****************************************************************************/
= 163 START:TASK;
= 164    CALL GETNAM;
= 165    PUT TO A1 BY SKIP;
= 166    CALL SCRSET (MNAME);       /* set up screen for drawing   */
= 167    CALL CLEAR;                /* clear this screen   */
= 168    CALL SCRCHG (MNAME);       /*  and display it   */
= 169    CALL WIDTH(IWIDTH,YWIDTH);/* get actual screen resolution   */
= 170    XLIMIT = IWIDTH - 10;      /* mouse limited to screen area   */
= 171    YLIMIT = YWIDTH - 10;
= 172    MI  = IWIDTH // 2;         /* position mouse at middle of screen   */
= 173    MY  = YWIDTH // 2;
= 174    MIO = MI;
= 175    MYO = MY;
= 176    CALL SPRITL(MI,MY,10,10,MNAME,ARROWS);
= 177    ENABLE MAUS;
= 178    ACTIVATE SCRUP;
= 179    WHEN MAUS ACTIVATE MOUSE;
= 180 END;  /* of task START */
= 181
= 182
```

in Zeile 150 mit dem Zeichnen des Pfeiles beschäftigt, und die Maus wird bewegt. Durch den damit verbundenen Interrupt wird die Task MOUSE lauffähig und bekommt aufgrund der höheren Priorität auch den Prozessor zugeteilt. Wie noch zu sehen ist, verändert MOUSE den Wert von MX und MY und wird fertig. Wenn jetzt SCRUP mit dem Zeichnen des Pfeiles fortfährt, geschieht dies an einer anderen Stelle, und man erhält zwei getrennte Pfeilteile.

Um ein solches Dilemma zu vermeiden, gibt es die Semaphor-Variablen. Da sie zur Koordination von Tasks dienen, müssen sie zwangsläufig immer auf Modulebene vereinbart werden (Zeile 42). Semaphor-Variable enthalten ganzzahlige, positive Werte, die um eins erniedrigt werden, wenn ein REQUEST auftritt, und mit jedem RELEASE wieder um eins erhöht werden.

Die Wirkung ergibt sich daraus, daß eine Task, die einen REQUEST enthält, suspendiert wird, wenn die Semaphore 0 enthält. In der Task-Statusanzeige unter RTOS (Befehl 'L') steht für eine dermaßen angehaltene Task als Status 'SEMA'. Macht nun eine andere Task ein RELEASE, so enthält die Semaphore 1: die Task höchster Priorität, die auf diese Semaphore wartete, wird lauffähig und blockiert nun ihrerseits mit dem REQUEST die Semaphore für andere Tasks. Damit als erste Semaphor-Operation ein erfolgreicher REQUEST möglich ist, wird die Semaphore in der Vereinbarung mit dem Wert 1 vorbesetzt.

In dem Programm sieht dieser Vorgang folgendermaßen aus: SCRUP erniedrigt mit dem REQUEST in Zeile 134 die Semaphore SCRSEM auf den Wert 0.

```
= 183 /#### SCSAVE ##########################################################/
= 184 /#  save picture on floppy                                            #/
= 185 /##########################################################################/
= 186 SCSAVE:TASK;
= 187    DCL DRIVE  FIXED;
= 188    DCL FNAME  CHAR(8);
= 189
= 190    CALL GETNAM;
= 191    PUT 'Drive: ' TO A1 BY SKIP,A;
= 192    GET DRIVE FROM A1 BY SKIP,F(1);
= 193    PUT 'File : ',MNAME TO A1 BY SKIP,A,A,(2)X;
= 194    FNAME = OPSTR(MNAME);
= 195    PUT TO A1 BY SKIP;
= 196    CALL SCSAVE(MNAME,DRIVE,FNAME);
= 197 END; /# of task SCSAVE #/
= 198
= 199 /#### SCLOAD #########################################################/
= 200 /#  load picture from floppy                                          #/
= 201 /##########################################################################/
= 202 SCLOAD:TASK;
= 203    DCL DRIVE  FIXED;
= 204    DCL FNAME  CHAR(8);
= 205
= 206    CALL GETNAM;
= 207    PUT 'Drive: ' TO A1 BY SKIP,A;
= 208    GET DRIVE FROM A1 BY SKIP,F(1);
= 209    PUT 'File : ',MNAME TO A1 BY SKIP,A,A,(2)X;
= 210    FNAME = OPSTR(MNAME);
= 211    PUT TO A1 BY SKIP;
= 212    CALL SCRSET(MNAME);
= 213    CALL SCLOAD(MNAME,DRIVE,FNAME);
= 214 END; /# of task SCLOAD #/
= 215
= 216
= 217 /#### CLEAR ##########################################################/
= 218 /#  clear screen                                                      #/
= 219 /##########################################################################/
= 220 CLEAR:TASK;
= 221    CALL GETNAM;
= 222    PUT TO A1 BY SKIP;
= 223    CALL SCRSET(MNAME);
= 224    CALL CLEAR;
= 225 END; /# of task CLEAR #/
= 226
= 227
= 228 MODEND;
```

Das Programmierbeispiel in dieser Folge unseres PEARL-Kurses ist ein Grundprogramm zum Zeichnen mit Mausunterstützung.

Wird nun die Maus bewegt, und die Task MOUSE möchte MX und MY ändern, so macht sie auch einen REQUEST auf SCRSEM (Zeile 100). Da diese aber den Wert 0 enthält, wird MOUSE suspendiert und SCRUP läuft weiter, bis sie in Zeile 154 SCRSEM wieder freigibt. Mit diesem RELEASE wird MOUSE lauffähig, der REQUEST in Zeile 100 ist erfolgreich, und MOUSE kann MX und MY ändern, ohne mit SCRUP in Konflikt zu geraten.

An dieser Stelle möchte ich auch auf eine Gefahr hinweisen, die sich aus Semaphoren ergeben kann. Wenn mit mehreren Semaphoren gearbeitet wird, kann sich die Situation ergeben, daß Task A durch eine von Task B belegte Semaphore blockiert wird, Task B diese aber nicht freigeben kann, weil sie ihrerseits auf eine von Task A belegte Semaphore wartet... Diese Situation wird anschaulich durch die Bezeichnung 'deadly embrace' ('tödliche Umarmung') beschrieben, da keine der beiden Tasks mehr lauffähig werden kann. Besonders heimtückisch sind solche Konstellationen dadurch, daß sie nicht einfach zu erkennen sind. Vielleicht treten sie nur unter seltenen zeitlichen Randbedingungen auf, die zufälligerweise während der Testphase eines Programmes nicht vorkommen, nach Murphy dann aber im Betrieb den größtmöglichen Schaden anrichten.

Interrupt als Wecker

In Zeile 155 wird schließlich eine weitere Einsatzmöglichkeit von Interrupts deutlich: als Endebedingung für ein RESUME. Auch hier ist ein Interrupt als Ersatz für ein zeitliches Ereignis einsetzbar, wie es in Teil 3 dieser

Reihe vorgestellt wurde. Somit wartet SCRUP jetzt auf den SYNC-Interrupt und muß auf Hilfe von außen hoffen, denn SYNC war ja von SCRUP in Zeile 132 gesperrt worden. Die Freigabe von SYNC geschieht in Zeile 119 durch die Task MOUSE, wenn diese neue Mauskoordinaten bereitgestellt hat und der Bildschirm somit zu aktualisieren ist. SCRUP läuft also nur, wenn dies durch Mausbewegungen notwendig ist, ansonsten ist sie suspendiert und verbraucht nicht unnütz Zeit.

Doch nun zu der Task MOUSE. Bei ihr fällt hinter der schon bekannten Prioritätsangabe das Wort RESIDENT auf. Bei Rechnern, die nicht laufende Tasks auf externe Speicher auslagern, würde dies fordern, daß die Task immer im Hauptspeicher bleiben soll, damit sie schnell aktivierbar ist. Da RTOS keine Tasks eigenständig auslagert, hat es hier nur den Effekt, daß der Task-Workspace nach Beendigung der Task nicht wieder freigegeben wird und bei der nächsten Aktivierung sofort zur Verfügung steht. Da MOUSE häufig aktiviert wird, führt dies zu einer merklichen Zeiteinsparung.

Bleibt nur noch zu erläutern, wie die Task MOUSE an die Mausdaten herankommt. Ich hatte schon oben erwähnt, daß die Mausdaten in einen Puffer geschrieben werden. Da die Adresse dieses Puffers bekannt ist, können diese Daten mit einer Konstruktion, die eigentlich für Peripherie-Bausteine gedacht ist, leicht gelesen werden. Dies beruht darauf, daß der 68000 keine eigenen Adressen für Ein- und Ausgabe hat und die Register von Peripherie-Bausteinen deshalb wie Speicherstellen angesprochen werden (im Fachchinesisch: 'memory mapped I/O'). Um auch von PEARL aus diese Register ansprechen zu können, kann man Speicheradressen als Datenstationen definieren und dann lesen und beschreiben.

Hinter diesen Speicheradressen muß sich dann allerdings nicht zwangsläufig ein Peripherie-Baustein verbergen, sondern man kann auf diese Weise auch normale Speicherplätze ansprechen. Wenn diese Speicherplätze wichtige Systemdaten enthalten, kann man auf diese Weise auch RTOS zum Absturz bringen. Solche gefährlichen Sachen will ich hier aber nicht vorführen, es ist nur als Warnung davor gedacht, auf unbekannte Speicherstellen schreibend zuzugreifen.

Vereinbart wird eine Speicherstelle als Datenstation im Systemteil durch Angabe eines Namens und hinter dem Doppelpunkt 'BU(hexnum)', wobei hexnum eine 6stellige Adresse in sedezimaler Form ist; davor steht hier noch 01 als Kennzeichen dafür, daß byteweise zugegriffen werden soll (Zeilen 12, 13, 14). In der Spezifikation im Problemteil (Zeilen 20, 21, 22) besagt BASIC, daß in diesem Falle nur jeweils ein Byte binärer Daten gelesen werden kann. Für das Lesen aus Datenstationen vom Typ BASIC gibt es einen eigenen Befehl, der 'TAKE' lautet (Zeilen 84, 85, 86), ebenso, wie nur mit 'SEND' in solche Datenstationen geschrieben werden kann.

Das Datenformat für die Mausdaten ist durch den Tastaturprozessor vorgegeben und daher identisch mit dem unter TOS. In dem ersten Byte sind die Bits 7 bis 3 gesetzt, Bit 2 zurückgesetzt, und Bit 0 und 1 enthalten die Information, ob der rechte und linke Mausknopf gedrückt sind. In den Zeilen 91 bis 98 wird diese Information in die Bitvariablen LBTN und RBTN übertragen. Byte 2 und 3 enthalten in einem vorzeichenbehafteten Byte die Mausbewegung in x- und y-Richtung. Da diese Bytes jeweils in die obere Hälfte der FIXED-Variablen gelesen werden, muß noch durch 256 geteilt werden (Zeilen 102, 111).

Bilder auf Diskette speichern

Leider stehen die Funktionen SCSAVE und SCLOAD, mit denen man die oben vorgestellten Bildschirmmodule auf Diskette speichern und von dort wieder laden kann, nicht von der Kommandoebene aus zur Verfügung. Deshalb habe ich sie durch gleichnamige Tasks zugänglich gemacht. Als Parameter werden der Name des Bildschirmmodules, die Laufwerksnummer (0: Diskettenlaufwerk 1, 1: Diskettenlaufwerk 2, 2: Festplattenpartition 1, . . .) sowie der Name, unter dem das Bildschirmmodul gespeichert wird, im Dialog eingegeben. Auf gleiche Weise ist ab Zeile 217 die Funktion CLEAR zum Löschen eines Bildschirms von der Kommandoebene aus aufrufbar.

Zu einem richtig komfortablen Zeichenprogramm fehlen dem vorgestellten Beispielprogramm natürlich noch ein paar Eigenschaften. So müßte beispielsweise beim Zeichnen und beim Löschen die Strichbreite frei wählbar sein – ein Feature, das Sie nach der oben schon erwähnten Methode jetzt gewiß leicht selbst hinzufügen können. Vielleicht reizt es Sie auch, Möglichkeiten zum Zeichnen mit der 'Gummiband-Methode' oder zum Herstellen von geometrischen Figuren vorzusehen. Nur zu, unter PEARL sind dem Programmierdrang kaum Grenzen gesetzt.

Assembler-Programmierung

Fühlen Sie sich auch nur dann wohl, wenn Sie den Takt im Prozessor schlagen hören und die Bits eigenhändig durch die Register schieben? Damit ist es auch unter einem echtzeitfähigen Multitasking-Betriebssystem nicht vorbei, wenngleich natürlich schon einige Regeln beachtet werden müssen. Aber das gilt schließlich für alle Betriebssysteme. Und manches läßt sich nun mal einfacher, eleganter und schneller in Assembler programmieren.

Eine erste Anwendung für Assembler-Programmierung sind Betriebssystemfunktionen, die dem PEARL-Programmierer nicht zugänglich sind, deren Leistungsfähigkeit aber reizvolle Möglichkeiten eröffnet. Zur Demonstration habe ich das in Teil 2 vorgestellte Adreßverwaltungsprogramm um eine dynamische Speicherverwaltung erweitert, die die bestehenden Grenzen des Wachstums der Datensätze sprengt.

In der ursprünglichen Version war die Datenmenge ja durch die Größe des Feldes bestimmt, in dem die Adreßstrukturen gespeichert wurden. Natürlich kann man in dem Programm ein großes Feld vorsehen, das so schnell nicht voll wird. Aber damit hätte man sowohl im Rechner als auch dann auf der Diskette unnötig viel Speicherkapazität belegt, solange das Feld noch weitgehend leer ist.

Außerdem entwickeln Dateien normalerweise eine unglaubliche Eigendynamik und wachsen dann sehr schnell. Demgegenüber wird jetzt nur soviel Speicher belegt, wie für die vorhandenen Datensätze tatsächlich benötigt wird.

Übrigens zwingt das Programm in der vorliegenden Variante den Anwender, nach Eingabe von 50 Adressen die Datei auf der Diskette zu sichern. Nach dem erneuten Laden steht jeweils wieder Platz für 50 weitere Einträge zur Verfügung.

Doch schön der Reihe nach. Wenn der Programmierer in einem PEARL-Programm ein modulglobales Feld vereinbart, berechnet der Compiler aus der Anzahl der Felder und der Länge der Objekte in den einzelnen Feldern den benötigten Speicherraum und reserviert ihn in dem Modul. Da die Adreßstruktur 102 Bytes groß ist, reservierte der Compiler für die 50 Feldelemente 5100 Bytes, weshalb ein entsprechend großer Wert in der 'S='-Angabe am Anfang des Programms erschien.

Dieses Feld habe ich ausgelagert, wodurch der durch das Programm fest belegte Speicher auf weniger als die Hälfte schrumfte. Allerdings steht nun kein Speicher mehr zur Verfügung, wenn das Programm gestartet ist. Der muß erst beim Betriebssystem angefordert werden. Dieses geschieht beim Laden einer schon bestehenden Datei oder aber durch den neu eingefügten Menüpunkt 'G: Adreßdatei anlegen' (Zeile 286, 343 - 353). Diesen Weg will ich im einzelnen verfolgen.

Vorher müssen Sie aber noch erfahren, wie ein Feld unter RTOS/PEARL aufgebaut ist.

Es besteht aus zwei Teilen: dem Feldbeschreibungsblock und dem Speicherbereich, in dem die Feldelemente tatsächlich abgelegt werden. Der Feldbeschreibungsblock beginnt mit bis zu drei Worten, in denen steht, wie groß das Feld in der jeweiligen Dimension ist. Deshalb können Felder auch nicht mehr als 32 767 Elemente enthalten. Darauf folgt ein Zeiger, der auf das 0-te Element des Feldes zeigt. Da für PEARL definiert ist, daß Felder mit dem Element 1 beginnen, zeigt dieser Zeiger also irgendwo vor das Feld. Das mag zwar merkwürdig erscheinen, macht aber die Berechnung der Adressen der einzelnen Feldelemente schneller.
Wenn das Feld von dem PEARL-Compiler eingerichtet worden ist, folgt auf diesen Zeiger der reservierte Speicherbereich. Das ist aber kein Muß, was ich ausnutze, indem ich den Zeiger nach Anforderung des nötigen Speichers auf diesen richte. Somit muß nur noch der Platz für den Feldbeschreibungsblock fest vereinbart werden. Dies geschieht in dem As-

sembler-Modul in den Zeilen 36 bis 38 für das Feld, das später die Strukturen enthält, und in den Zeilen 39, 40 für das Feld, das zum Sortieren gebraucht wird. Dabei bedeutet die nach links geöffnete spitze Klammer, daß es sich um ein globales Symbol handelt, auf das von dem PEARL-Modul aus zugegriffen werden kann. Da die beiden Felder eindimensional sind, ist jeweils ein Wort nötig für die Feldgröße, hinter denen ein Langwort folgt für den Zeiger.

Besondere Beachtung verdient Zeile 37, denn durch sie wird die Feldverwaltung erst so richtig dynamisch. Normalerweise ist der Feldbeschreibungsblock ja für die Programme nicht greifbar und somit die Feldgröße unveränderlich. Durch das Spezifizieren der Feldgröße als Variable aber ist sie für das Programm greifbar. Wird eine Adresse eingegeben oder gelöscht, so wächst beziehungsweise schrumft das Feld für alle Funktionen, die auf es zurückgreifen, da NOOADR als Zähler für die verfügbaren Adressen geführt wird. Das ist besonders für die Funktion WRITE wichtig, da diese über die Felddimensionsangaben die Anzahl der auf die Diskette zu schreibenden Bytes berechnet, und somit keine 'leeren' Adressen wertvollen Diskettenplatz belegen.

Ein weiterer Seitensprung ist nötig. Auf PEARL-Ebene muß man sich ja auch um die Parameterübergabe an eine Funktion oder Prozedur nicht weiter kümmern, das erledigt der Compiler. Für den Assembler-Programmierer aber sind die genauen Spielregeln wichtig.

```
*   1 000000                  ****************************************
*   2 000000                  *                                      *
*   3 000000                  *  PROCEDUREN ZUR DYNAMISCHEN SPEICHERVERWALTUNG
*   4 000000                  *   MIT DEM ADRESSVERWALTUNGSPROGRAMM 'DB'
*   5 000000                  *                                      *
*   6 000000                  * last update:   11.09.86        15:00
*   7 000000                  *                                      *
*   8 000000                  ****************************************
*   9 000000                  *       MODULE HEAD FOR RTOS-UM        *
*  10 000000                  *                                      *
*  11 000000 00000000                DC.L  0                 for loader
*  12 000004 00000000                DC.L  0                 for loader
*  13 000008 0010                    DC    $10               type: MOLE
*  14 00000A 444253202020            DC.B  'DBS              module name
*  15 000010                  ****************************************
*  16 000010                  *       SYSTEM TRAPS                   *
*  17 000010                  *                                      *
*  18 000010                  RETN  OPD   $4E4C           return from proc.
*  19 000010                  OFF   OPD   $4E4F           interrupts off
*  20 000010                  WSBS  OPD   $A0XC           wsp backward search
*  21 000010                  RWSP  OPD   $A0CA           release workspace
*  22 000010                  EPAR  OPD.V 19              end of parm offer
*  23 000010                  ENTP  OPD.V 29              procedure entry
*  24 000010                  LTST  OPD.V 72              limitest struc offer
*  25 000010                  ARNS  OPD.V 122             prm offer array
*  26 000010                  ****************************************
*  27 000010                  *       MEMORY SECTION DEFINITIONS     *
*  28 000010                  *                                      *
*  29 000010 0000000B         TYPE  EQU   8               type of mem section
*  30 000010 0000000A         NAME  EQU   10              name of mem section
*  31 000010 0000000A         FORT  EQU   10              forward T-link
*  32 000010 0000000E         BACKT EQU   14              backward T-link
*  33 000010                  ****************************************
*  34 000010                  *       ARRAY HEADER OF GLOBAL ARRAYS  *
*  35 000010                  *                                      *
*  36 000010                  DATE1: DS   0
*  37 000010 0000             NOOADR: DC  0               dimension of array
*  38 000012 00000000                 DC.L 0               ptr to 0th element
*  39 000016 0000             ORDER: DC   0               dimension of array
*  40 00001B 00000000                 DC.L 0               ptr to 0th element
*  41 00001C 00000000         MEMBLK: DC.L 0              ptr to allocated mem
*  42 000020                  ****************************************
*  43 000020                  *       PROCEDURE WORKSPACE DEFINITIONS*
*  44 000020                  *                                      *
*  45 000020 00000000         BND0  EQU   0               array 1st dimension
*  46 000020 00000002         OFF0  EQU   BND0+2          array offset
*  47 000020 00000006         BND1  EQU   OFF0+4          array 1st dimension
*  48 000020 0000000B         OFF1  EQU   BND1+2          array offset
*  49 000020 0000000C         WSPSZ EQU   OFF1+4          proc workspace size
*  50 000020                  ****************************************
*  51 000020                  *       CONSTANTS                      *
*  52 000020                  *                                      *
*  53 000020 00000002         WRLEN EQU   2               length of FIXED(15)
*  54 000020 00000066         STCLEN EQU  $66             length of ADDRESS
*  55 000020                  ****************************************
*  56 000020                  *       ALLOCATE MEMORY FOR DATA ARRAYS*
*  57 000020                  ****************************************
*  58 000020                  *       REGISTER USAGE                 *
*  59 000020                  *       A2: address of DATE1           *
*  60 000020                  *       A3: address of ORDER           *
*  61 000020                  *                                      *
*  62 000020 1DC000000000     ALLOC: ENTR  WSPSZ.L            fetch storage
*  63 000026 7A0000020000
*            0415                     ARNS  OFF0.X,BND0.X,$0415.X  array offer
```

Besondere Beachtung verdient Zeile 37, denn durch sie wird die Feldverwaltung erst so richtig dynamisch. Normalerweise ist der Feldbeschreibungsblock ja für die Programme nicht greifbar und somit die Feldgröße unveränderlich. Durch das Spezifizieren der Feldgröße als Variable aber ist sie für das Programm greifbar. Wird eine Adresse eingegeben oder gelöscht, so wächst beziehungsweise schrumft das Feld für alle Funktionen, die auf es zurückgreifen, da NOOADR als Zähler für die verfügbaren Adressen geführt wird. Das ist besonders für die Funktion WRITE wichtig, da diese über die Felddimensionsangaben die Anzahl der auf die Diskette zu schreibenden Bytes berechnet, und somit keine 'leeren' Adressen wertvollen Diskettenplatz belegen.

Ein weiterer Seitensprung ist nötig. Auf PEARL-Ebene muß man sich ja auch um die Parameterübergabe an eine Funktion oder Prozedur nicht weiter kümmern, das erledigt der Compiler. Für den Assembler-Programmierer aber sind die genauen Spielregeln wichtig.

Hyperprozessor

Mit RTOS läuft ein weiterer Prozessor auf Ihrem Rechner, der sogenannte 'Hyperprozessor'. Wer vom APPLE II noch 'SWEET16' kennt und hier eine Parallele vermutet, ist schon auf dem richtigen Wege. SWEET16 wurde geboren aus der Not, viele 16-Bit-Operationen auf dem 8-Bit-Prozessor 6502 durchführen zu müssen. Ähnlich hier, wenngleich der 68000-Programmierer natürlich über die Probleme eines 6502 mit 16-Bit-Operationen nur müde lächeln kann. Im Vergleich aber etwa mit den Assembler-Befehlen einer IBM 370 stellt auch der 68000 nur sehr einfache Befehle zur Verfügung. Für kompliziertere Operationen gibt es deshalb unter RTOS den Hyperprozessor. Dabei handelt es sich um einen Programmteil im Betriebssystem, der die komplexen 'Hyperproc-Befehle' interpretiert. Der Hyperprozessor

```
*  64 000002E 4B0000b6          LTST   STCLEN.X        to check struc len   *
*  65 0000032 7A0000008000b
*             0401                ARNS   OFF1.X,BND1.X,$0401.X  array xfer   *
*  66 0000036 1700                EPAR                end of parm xfer     *
*  67 000003C 227AFFD2            MOVE   NODADR,D1    get value of MAXADR  *
*  68 0000040 45FAFFDE            LEA    DATEI,A2     get address of DATEI *
*  69 0000044 47FAFFD0            LEA    ORDER,A7     get address of ORDER *
*  70 000004B                 *  reserve memory for 50 additional addresses *
*  71 000004B 227C00032           ADD    #50,D1       reserve more memory  *
*  72 000040 7e91                 MOVE   D1,(A7)      set bound of ORDER   *
*  73 000040E 7001                MOVE   D1,D0        get a copy           *
*  74 0000050 C3FC0002            MULS   #WORDLEN,D1  memory for ORDER     *
*  75 0000054 C1FC00bb            MULS   #STCLEN,D0   memory for DATEI     *
*  76 000005B 6B7a                BMI.S  NOMEM        bi NODADR is negativ *
*  77 000005A D2B0                ADD.L  D0,D1        needed memory        *
*  78 000005C D2BC0000002D        ADD.L  #$20,D1      for mem section head *
*  79 0000b2 A00C                 WBFS                allocate memory      *
*  80 0000b4 6b2A                 BNE.S  NOMEM        bi not enough memory *
*  81 0000b6 41FAFFB4             LEA    MEMBLK,A0    get address of MEMBLK *
*  82 0000bA 20B9                 MOVE.L A1,(A0)      save adr of mem sect *
*  83 0000bC D3FC0000002D         ADDA.L #$20,A1      calc start address   *
*  84 0000C2 93FC0000006b         SUBA.L #STCLEN,A1    of 0th element of   *
*  85 0000C8 2549000C             MOVE.L A1,2(A2)      DATEI               *
*  86 0000CE D3FC0000006b         ADDA.L #STCLEN,A1
*  87 0000D2 D7C0                 ADDA.L D0,A1        calc start address   *
*  88 0000D4 93FC0000000C         SUBA.L #WORDLEN,A1   of 0th element of   *
*  89 0000DA 2749000C             MOVE.L A1,2(A7)      ORDER               *
*  90 0000DE 4E4C                 RETN                back to caller       *
*  91 0000E0                  *
*  92 0000E0 7200             NOMEM: MOVEQ  #0,D1        to indicate error    *
*  93 0000E2 7481                 MOVE   D1,(A2)      reset bound of DATEI *
*  94 0000E4 7e81                 MOVE   D1,(A7)      reset bound of ORDER *
*  95 0000E6 4E4C                 RETN                back to caller       *
*  96 0000E8 ******************************************************************
*  97 0000E8 *          FREE ALLOCATED MEMORY                                 *
*  98 0000E8 ******************************************************************
*  99 0000E8 0000             FREE:  VO                  hyperprocessor off   *
* 100 0000EA 47FAFFB0             LEA    MEMBLK,A0    get address of MEMBLK *
* 101 0000EE 4A91                 TST.L  (A1)         memory allocated?    *
* 102 0000F0 6742                 BEQ.S  FREEIT       bi no memory allocated*
* 103 0000F2 4E4F                 OFF                 no irq allowed       *
* 104 0000F4                 *    unlink memory section from T-link        *
* 105 0000F4 227AFFF6             MOVE.L MEMBLK,A1    get address of mem block *
* 106 0000F8 24a9000E             MOVE.L (A0)T(A1),A2  adr of left neighbor *
* 107 0000FC 25A9000400A0A        MOVE.L FORT(A1),FORT(A2) forward T-link  *
* 108 0000B2 24a9000A             MOVE.L FORT(A1),A2   adr of right neighbor *
* 109 0000B6 25A9000E000E         MOVE.L BACKT(A1),BACKT(A2) backward T-link *
* 110 0000BC 227C00100008         MOVE   #$10,TYPE(A1) declare as module    *
* 111 0000C2 227C41420004         MOVE   #$4142,NAME(A1) name module        *
* 112 0000C8 227C47444000         MOVE   #$4744,NAME+2(A1)                  *
* 113 0000CE 227C4546000E         MOVE   #$4546,NAME+4(A1)                  *
* 114 0000D4 027C0BFF             ANDI   #$0BFF,SR    enable interrupts    *
* 115 0000D8 A00A                 RWSF                free memory section  *
* 116 0000DA 42FAFF40             LEA    MEMBLK,A1    get address of MEMBLK *
* 117 0000DE 22BC00000000         MOVE.L #0,(A1)      no memory allocated  *
* 118 0000E4 4EEB0002     FREEIT: JMP    2(A0)        back to caller       *
* 119 0000EB ******************************************************************
* 120 0000EB 60000000             END    OF                JOY              *
```

Das Assembler-Modul verleiht dem Adreßverwaltungsprogramm die Fähigkeit zur dynamischen Speicherverwaltung.

besitzt so etwas wie einen Program Counter, mit dem er seine Befehle liest, um sie dann softwaremäßig zu dekodieren und abzuarbeiten. Dies entspricht, wenn man so will, der Arbeitsweise eines BASIC-Interpreters, nur wird dort mit Zeilennummern gearbeitet.

Das Geschäft der Parameterübergabe übernimmt also der Hyperprozessor ebenso wie den Prozeduraufruf und die Workspace-Anforderung. Denn genau dieses tut der erste Befehl in der Prozedur ALLOC (Zeile 62). Dies ist ein wichtiger Punkt in bezug auf die Wiedereintrittsfähigkeit der Prozedur. Da mit jedem Prozeduraufruf ein neuer Speicherbereich als Workspace angefordert wird, können mehrere Tasks gleichzeitig mit der Prozedur arbeiten, ohne daß die Parameter durcheinander geraten. Der Preis, den wir hierfür zahlen müssen, besteht darin, daß auf den Workspace nur indirekt zugegriffen werden kann. Dazu wird vom Betriebssystem ein Zeiger auf den Workspace in A5 bereitgestellt. Da variable(A5) viel Schreibarbeit bedeuten würde, kann man auch statt dessen variable.X schreiben.

In den nächsten drei Assembler-Zeilen werden die Parameter in Empfang genommen, die mit CALL ALLOC(DATEI, ORDER) auf PEARL-Ebene abgesandt worden waren. Dabei werden mit OFFx.X und BNDx.X die Werte des Feldbeschreibungsblockes in den Prozedur-Workspace übertragen, während $0415 und LTST STCLEN.X die Informationen über Art des Feldinhaltes (hier Strukturen) und Größe der Feldelemente ($66 = 102, siehe oben) enthalten. Daß ORDER FIXED15-Werte enthält, läßt sich aus $0401.X entnehmen, und da FIXED15-Werte immer 2 Byte lang sind, ist kein LTST-Befehl (length test) erforderlich.

EPAR schließt die Parameterübergabe ab und schaltet gleichzeitig den Hyperprozessor aus, so daß man in ganz normalem 68000-Code weiterschreiben kann. Diese Feldübergabe ist eine reine Vorsichtsmaßnahme, denn über die Felder ist ja in dem Assembler-Modul alles bekannt. Es wird aber von dem Hyperprozessor beim Parametertransfer noch überprüft, ob die Datentypen und Längen übereinstimmen. So kommt es beispielweise zu einer Fehler-

```
  1 /*****************************************************************/
  2 /*                                                               */
  3 /*   A D D R E S S   D A T A B A S E                             */
  4 /*   last update:  11.09.86     14h45                            */
  5 /*                                                               */
  6 /*   version 1.2 for dynamic memory allocation                   */
  7 /*                                                               */
  8 /*****************************************************************/cm*/
  9 S=1220;
 10 MODULE DB;
 11 SYSTEM;
 12    A1;                    /* keyboard, screen */
 13    X1; A1(TFU=1);         /* for input of just 1 char */
 14    PP;                    /* centronics parallel port */
 15    DF; F0,DATEI;          /* address file */
 16
 17 PROBLEM;
 18    SPC A1 DATION INOUT ALPHIC CONTROL(ALL);
 19    SPC X1 DATION INOUT ALPHIC CONTROL(ALL);
 20    SPC PP DATION   OUT ALPHIC CONTROL(ALL);
 21    SPC DF DATION INOUT ALPHIC CONTROL(ALL);
 22
 23 /* structure definitions */
 24    TYPE ADDRESS STRUCT (/NAME     CHAR(20),
 25                         VORNAME CHAR(20),
 26                         STRASSE CHAR(20),
 27                         PLZ     FIXED,
 28                         ORT     CHAR(20),
 29                         TELEFON CHAR(20) /);
 30
 31 /* binding of global functions */
 32    SPC (WRITE,READ) ENTRY GLOBAL;
 33    SPC ALLOC        ENTRY GLOBAL;
 34    SPC FREE         ENTRY GLOBAL;
 35
 36
 37 /* specification of global variables */
 38    SPC DATEI() ADDRESS GLOBAL;
 39    SPC ORDER() FIXED   GLOBAL;
 40    SPC NOOADR  FIXED   GLOBAL;
 41
 42 /* constant declarations */
 43    DCL TRUE   INV BIT(1) INIT ('1'B);
 44    DCL FALSE  INV BIT(1) INIT ('0'B);
 45    DCL ADROFF INV FIXED  INIT (50);
 46
 47 /* variables declaration */
 48    DCL MAXADR FIXED;
 49
 50


198 /**** LOAD ***************************************************/
199 /* hole Dateidaten von Diskette                               */
200 /*****************************************************************/
201 LOAD:PROC;
202
203    OPEN DF;
204    CALL REWIND (DF);
205    CALL READ (DF,NOOADR);
206    CALL FREE;
207    CALL ALLOC (DATEI, ORDER);
208    IF NOOADR EQ 0
209       THEN PUT 'ERROR: NOT ENOUGH MEMORY' TO A1 BY SKIP,A,SKIP;
210            NOOADR = 0;
211            MAXADR = 0;
212            RETURN;          /* exit of procedure */
213    FIN;
214    CALL READ (DF,DATEI);
215    CLOSE DF;
216    MAXADR = NOOADR + ADROFF;
217 END; /* of procedure LOAD */
218
219
220 /**** SAVE ***************************************************/
221 /* speichere Datei Inhalt auf Diskette                        */
222 /*****************************************************************/
223 SAVE:PROC;
224    DCL DUMMY INV FIXED INIT(1234);
225
226    IF NOOADR EQ 0
227       THEN PUT 'keine Adressen vorhanden zum Speichern'
228            TO A1 BY (2)SKIP,A,SKIP;
229            RETURN;          /* exit procedure */
```

meldung, wenn nur in dem PEARL-Programm, nicht aber auf Assembler-Seite, etwas an der Adreßstruktur geändert wird. Sonst könnte eine solche Änderung nämlich zu einem Betriebssystemabsturz führen, dem eventuell nur schwer auf die Schliche zu kommen ist.

Bis Zeile 78 wird dann der Speicherbedarf für die beiden Felder zuzüglich 50 Adressen Reserve und $20 Bytes Verwaltungskopf des zu reservierenden Speicherblocks berechnet. Mit diesem Wert in D1 wird der Line-A-Trap WSBS aufgerufen, der bei den hohen Adressen zu suchen beginnt, bis er einen Speicherblock genügender Größe gefunden hat. Dessen Adresse wird in A1 zurückgegeben. Ist aber das Zero Flag bei der Rückkehr zurückgesetzt, dann ist kein geeigneter Speicherblock gefunden worden. Eine 0 in NOOADR macht diesen unglücklichen Umstand sodann der PEARL-Welt bekannt. Wurde aber ein Speicherblock gefunden, der groß genug ist, werden die beiden Zeiger in den Feldbeschreibungsblöcken so gesetzt, daß die Daten in dem Speicherblock landen, auch wenn der Zeiger von DATEI vor den Speicherblock zeigt.

Genau das Gegenteil wird mit der Funktion FREE erreicht: damit wird der mit ALLOC angeforderte Speicherblock zurückgegeben. Dazu wird die Startadresse des Speicherblocks benötigt, die dafür in Zeile 81, 82 in MEMBLK gespeichert wird. FREE überprüft in Zeile 100 bis 102, ob Speicher angefordert wurde, und kehrt sofort zum Aufrufer zurück, wenn dies nicht der Fall ist. Ansonsten schaltet die Routine mit dem Trap 15 (OFF) alle Interrupts ab und löst den Speicherblock aus der T-link-Kette.

```
 230    FIN;
 231    OPEN DF;
 232    CALL REWIND (DF);
 233    CALL WRITE (DF,NOOADR,DATEI,DUMMY);
 234    CLOSE DF;
 235    CALL FREE;     /* free allocated memory */
 236    NOOADR = 0;
 237    MAXADR = 0;
 238 END; /* of procedure SAVE */
 239
 240


 262 /**** DB ****************************************************************/
 263 /* Adress Verwaltungs Programm                                          */
 264 /*********************************************************************/
 265 DB:TASK;
 266    DCL DONE  BIT(1) INIT('0'B);
 267    DCL EXIT  BIT(1);
 268    DCL INDEX FIXED;
 269    DCL NAME  CHAR(20);
 270    DCL FRAGE CHAR(32);
 271    DCL CHR   CHAR(1);
 272
 273    NOOADR = 0;        /* noch keine Adressen verfuegbar */
 274    MAXADR = 0;        /* noch keine Adressen verfuegbar */
 275    WHILE NOT DONE REPEAT   /* Kommando Schleife */
 276'      /* Gib Menue aus */
 277       PUT 'Adressen Verwaltung' TO A1 BY (5)SKIP,A;
 278       PUT '    Adressen: ',NOOADR TO A1 BY A,F(5);
 279       PUT '    Frei: ',MAXADR - NOOADR TO A1 BY A,F(5),(2)SKIP;
 280       PUT 'A: Adressen zeigen' TO A1 BY SKIP,A;
 281       PUT 'B: Adresse suchen' TO A1 BY SKIP,A;
 282       PUT 'C: Adresse eingeben' TO A1 BY SKIP,A;
 283       PUT 'D: Adresse loeschen' TO A1 BY SKIP,A;
 284       PUT 'E: Adressdatei lesen' TO A1 BY SKIP,A;
 285       PUT 'F: Adressdatei speichern' TO A1 BY SKIP,A;
 286       PUT 'G: Adressdatei anlegen' TO A1 BY SKIP,A;
 287       PUT 'H: Ausgang' TO A1 BY (2)SKIP,A;
 288       PUT 'Wahl: ' TO A1 BY (2)SKIP,A;
 289       /* Hole Kommando */
 290       GET CHR FROM X1 BY SKIP,A;
 291       /* Fuehre Kommando aus */
 292       EXIT = FALSE;
 293       CASE TOFIXED(LTOU(CHR)) - TOFIXED('A') + 1
 294
 295          ALT /* A */   /* Adressen zeigen */
 296             CALL SORT;
 297             FRAGE = 'Weiter (''J'',''E''xit,''D''rucken): ';
 298             FOR I TO NOOADR WHILE NOT EXIT REPEAT
 299                CHR = SELECT (DATEI(ORDER(I)),FRAGE,EXIT);
 300             END;
 301
 302          ALT /* B */   /* Adressen suchen */
 303             IF NOOADR EQ 0
 304                THEN PUT 'Keine Adressen vorhanden.'
 305                     TO A1 BY (2)SKIP,A,SKIP;
 306                ELSE PUT 'Name: ' TO A1 BY (2)SKIP,A;
 307                     GET NAME FROM A1 BY SKIP,A;
 308                     FRAGE = 'Weiter (''J'',''E''xit,''D''rucken): ';
 309                     INDEX = 1;
 310                     WHILE NOT EXIT REPEAT
 311                        INDEX = SEARCH(NAME,INDEX);
 312                        IF INDEX EQ 0
 313                           THEN EXIT = TRUE;
 314                           ELSE CHR = SELECT (DATEI(INDEX),FRAGE,EXIT);
 315                        FIN;
 316                        INDEX = INDEX + 1;
 317                     END; /* of WHILE loop */
 318             FIN;
 319
 320          ALT /* C */   /* Adresse eingeben */
 321             NOOADR = NOOADR + 1;
 322             IF NOOADR GT MAXADR
 323                THEN PUT 'Datei voll' TO A1 BY (2)SKIP,A,SKIP;
 324                     NOOADR = NOOADR - 1;
 325                ELSE DATEI(NOOADR) = GETADR;
 326             FIN;
 327
 328          ALT /* D */   /* Adresse loeschen */
 329             FRAGE = 'Loeschen (''J'',''N''ein,''E''xit): ';
 330             FOR I TO NOOADR WHILE NOT EXIT REPEAT
 331                IF 'J' EQ SELECT (DATEI(I),FRAGE,EXIT)
 332                   THEN DATEI(I) = DATEI(NOOADR);
 333                        NOOADR = NOOADR - 1;
 334                FIN;
 335             END;
```

Diese Kette besteht aus Vorwärts- und Rückwärtszeigern, mit deren Hilfe alle Speicherbereiche, die durch eine Task oder eine von ihr aufgerufene Prozedur angefordert wurden, bei einem UNLOAD task wieder freigegeben werden. Würde jetzt einfach nur der Speicherbereich mit RWSP (Zeile 115) freigegeben, so wäre die Kette unterbrochen, was zu einem Betriebssystem-Absturz führen kann. Da in dieses 'Auslinken' niemand hineinpfuschen darf, mußte mit OFF jede Unterbrechung untersagt werden, bis mit Zeile 114 die Interrupts wieder freigegeben werden.

Da FREE keinen Prozedur-Workspace benötigt, kann sie sofort mit dem Ausschalten des Hyperprozessors beginnen und auch über A0 zurückspringen, in dem die Adresse des Aufrufs enthalten ist. Nach 2(A0) muß gesprungen werden, da hinter dem Prozeduraufruf noch das EPAR für das Ende der leeren Parametertransfer-Liste steht. ALLOC dagegen muß mit RETN beendet werden, da ALLOC den Prozedur-Workspace auslinkt und wieder freigibt.

Die notwendigen Änderungen in dem PEARL-Programm halten sich in Grenzen. Zum einen sind natürlich im Kopf die externen Variablen und Prozeduren zu spezifizieren, zum andern in den Funktionen LOAD und SAVE die Prozeduraufrufe für ALLOC und FREE einzufügen. Schließlich ist noch im Hauptprogramm 'DB' die Funktion 'Datei anlegen' zu ergänzen, damit eine Datei neu eingerichtet werden kann.

Das assemblierte Modul wird durch den Lader/Linker mit dem kompilierten 'DB'-Programm verbunden, wenn man beim Aufruf in der Parameter-

```
*  336
*  337          ALT /* E */   /* Adressdatei von Diskette lesen */
*  338             CALL LOAD;
*  339
*  340          ALT /* F */   /* Adressdatei auf Diskette schreiben */
*  341             CALL SAVE;
*  342
*  343          ALT /* G */   /* Adressdatei einrichten */
*  344             NOOADR = ADROFF;
*  345             CALL ALLOC (DATEI, ORDER);
*  346             IF NOOADR EQ 0
*  347                THEN PUT 'ERROR: NOT ENOUGH MEMORY'
*  348                      TO A1 BY (2)SKIP,A,SKIP;
*  349                MAYADR = 0;
*  350             ELSE NOOADR = 0;
*  351                MAYADR = ADROFF;
*  352          FIN;
*  353
*  354          ALT /* H */   /* Programm beenden */
*  355             DONE = TRUE;
*  356             PUT 'Bye' TO A1 BY (2)SKIP,A,SKIP;
*  357
*  358          OUT /* invalid command */
*  359             ;  /* do nothing */
*  360
*  361       FIN; /* of CASE */
*  362    END; /* of REPEAT loop */
*  363 END; /* of task DB */
*  364
*  365 /*****************************************************************/
*  366 MODEND;
```

Diese Teile des Adreßverwaltungsprogramms 'DB' aus c't 9/86 müssen geändert oder ergänzt werden; die übrigen werden unverändert übernommen.

```
UH-ASMBLF-A15
*    1 000000                 ***************************************************
*    2 000000                 *        A S S E M B L E R   M O U S E            *
*    3 000000                 ***************************************************
*    4 000000                 *      module head for RTOS-UH                    *
*    5 000000 F0000001A       MDLE:  DC.L   TASK1          for the loader        *
*    6 000004 00000000        DC.L   0                     for the loader        *
*    7 000008 0010            DC     $10                                        *
*    8 00000A 415340445553    DC.B   'ASMOUS'                                   *
*    9 000010                 ***************************************************
*   10 000010                 *      hyperprocessor and trap definitions        *
*   11 000010                 TERMI  OPD    $4E41          terminate self        *
*   12 000010                 REQU   OPD    $4E46          request semaphore     *
*   13 000010                 RELEA  OPD    $4E47          release semaphore     *
*   14 000010                 ENAB   OPD    $A072          enable interrupt      *
*   15 000010                 DISAB  OPD    $A074          disable interrupt     *
*   16 000010                 ***************************************************
*   17 000010                 *      module variables                           *
*   18 000010 40000000        SYNC   DC.L   $40000000      vsync interrupt       *
*   19 000014 80000000        MAUS   DC.L   $80000000      mouse interrupt       *
*   20 00001B 0000            MTO    DC     0               old mouse status      *
*   21 00001A                 ***************************************************
*   22 00001A                 *                                                  *
*   23 00001A                 *      M O U S E                                   *
*   24 00001A                 *      last update:   02.09.86   20h55             *
*   25 00001A                 *                                                  *
*   26 00001A                 ***************************************************
*   27 00001A                 *      task head for RTOS-UH                      *
*   28 00001A 00000000        TASK1: DC.L   0                     for loader    *
*   29 00001E F0000000        DC.L   MDLE                         for loader    *
*   30 000022 00B1            DC     $B1                          type: resident task *
*   31 000024 424F55534520    DC.B   'MOUSE'                      task name     *
*   32 000028 000F            DC     $F10                         task priority *
*   33 00002C 000000FE        DC.L   WSPLEN                       work space length *
*   34 000030 0000000000000
*            0000             DC.L   0,0                          for System    *
*   35 000038 000F            DC     $F10                         also priority *
*   36 00003A F0000066        DC.L   MOUSE                        start-PC      *
```

liste die Namen der S-Records einfach durch ein '+' miteinander verknüpft. Also etwa:

LOAD DBSR+DBSSR

(vorausgesetzt, Sie haben beim Compiler- beziehungsweise Assembleraufruf diese Namen für die Zieldateien gewählt.)

Tasks auch in Assembler

Aber nicht nur kleine Unterprogramme lassen sich in Assembler schreiben. Als Demonstration dafür habe ich die in Teil 3 abgedruckte MOUSE-Task noch einmal in Assembler geschrieben.

Das Programm beginnt mit einem Modulkopf, wie er auch schon über den Prozeduren ALLOC und FREE auftauchte. Dieser Kopf wird von dem Betriebssystem für die Speicherverwaltung gebraucht. So werden in den ersten beiden Langwörtern je ein Vorwärts- und ein Rückwärtszeiger auf den nächsten Speicherblock abgelegt. In einem Modul sind die modulglobalen Variablen sowie die Prozeduren und Funktionen enthalten. In diesem Fall enthält das Modul zwei Konstantenwerte, nämlich die Bitmuster für den MAUS- und den SYNC-Interrupt (Zeile 18 bis 20) sowie Platz für ein Wort, in dem später der vom Keyboard übergebene Maus-Status gespeichert wird.

Etwas länger als der Modulkopf ist der Task-Kopf in Zeile 28 bis 37. Zusätzlich zu einem Modulkopf enthält er noch die Priorität der Task, die Länge des Task-Workspaces, einen Zeiger auf den Task-Beginn sowie Platz für verschiedene Variable, die später vom Betriebssystem benötigt werden.

```
 *  37 0000BE 00000000000000
 *            00000000000000
 *            00000000000000
 *            00000000000000
 *            00000000000000
 *            00000000000000
 *            00000000000000
 *  38 0000be                    DC.L   0,0,0,0,0,0,0,0,0,0  40 bytes zero   *
 *  39 0000be                 ***********************************************
 *  40 0000be 000000F         PRIO    EQU    15                 task priority   *
 *  41 0000be 00000FE         WSPLEN  EQU    $FE                workspace length *
 *  42 0000be                 ***********************************************
 *  43 0000be                 *       task workspace definitions              *
 *  44 0000be 00000FE         STACK   EQU    WSPLEN             workspace length *
 *  45 0000be                 ***********************************************
 *  46 0000be 4FEC00FE        MOUSE   LEA    STACK,T,A7         set up stack     *
 *  47 0000C4 203CFFA8                MOVE.L MAUS,D0            get interrupt mask *
 *  48 0000C8 A074                    DISAB                     disable MAUS interrupt*
 *  49 0000CA 103B00E6                MOVE.B $E6,D0             get new mouse status *
 *  50 0000CE B03CFFA2                CMP.B  MTO,D0             compare with old one *
 *  51 0000D2 671C                    BEQ.S  MM0                bz: didn't change  *
 *  52 0000D4 43FAFF9C                LEA    MTO,A1             get address of MTO *
 *  53 0000D8 1280                    MOVE.B D0,(A1)            save status        *
 *  54 0000DA 0800000001              BTST   #1,D0              test left button   *
 *  55 0000DA 66FF00CLBTN..           BNE    .LBTN              save result of test *
 *  56 0000DE 0800000                 BTST   #0,D0              test right button  *
 *  57 0000DE 66FF00CRBTN..           BNE    .RBTN              save result of test *
 *  58 0000E4 60BE                    BRA.S  MEXIT              it's done, go to exit *
 *  59 0000E6                 *       mouse has been moved                    *
 *  60 0000E6 42FF00SCRSEM MM0:       LEA    SCRSEM,A1          get address of sema *
 *  61 0000EC 4E46                    REQU                      request the semaphor *
 *  62 0000EE 103B00E7                MOVE.B $E7,D0             get x-movement     *
 *  63 0000F2 671E                    BEQ.S  MM4                bz: no x-movement  *
 *  64 0000F4 4880                    EXT.W  D0                 sign extend to word *
 *  65 0000F6 D07B00MX.... MM2:       ADD.W  MX,D0              add old position   *
 *  66 0000FC 0C7C0280                CMP.W  #$280,D0           test for screen border*
 *  67 0000FE 6C04                    BGE.S  MM3                bz: not out of range *
 *  68 000100 6000                    BRA.L  MM1                bz: x greater than 629 *
 *  69 000104                 *       x is negativ, replace by 0              *
 *  70 000104 7000                    MOVEQ  #0,D0              x = 0              *
 *  71 000106 6004                    BRA.S  MM2                                   *
 *  72 000108                 *       x is greater than 629, replace by 629   *
 *  73 000108 203C0275        MM1:    MOVE.W #$275,D0           x = 629            *
 *  74 00010C 33C00MX.... MM3:        MOVE.W D0,MX              save new x         *
 *  75 000112                 *       handle y-movement                       *
 *  76 000112 103B00E8        MM4:    MOVE.B $E8,D0             get y-movement     *
 *  77 000116 671E                    BEQ.S  MM5                bz: no y-movement  *
 *  78 000118 4880                    EXT.W  D0                 sign extend to word *
 *  79 00011A D07B00MY.... MM6:       ADD.W  MY,D0              add old position   *
 *  80 000120 0C7C01B8                CMP.W  #$1B8,D0           test for screen border*
 *  81 000124 6504                    BCS.S  MM8                bz: not out of range *
 *  82 000126 6A04                    BPL.S  MM7                bz: y greater than 389 *
 *  83 000128                 *       y is negativ, replace by 0              *
 *  84 000128 7000                    MOVEQ  #0,D0              y = 0              *
 *  85 00012A 6004                    BRA.S  MM9                                   *
 *  86 00012C                 *       y is greater than 389, replace by 389   *
 *  87 00012C 203C01B5        MM7:    MOVE.W #$1B5,D0           y = 389            *
 *  88 000130 33C00MY.... MM8:        MOVE.W D0,MY              save new y         *
 *  89 000136                 *       enable the SYNC interrupt               *
 *  90 000136 203AFF2B        MM9:    MOVE.L SYNC,D0            get interrupt mask *
 *  91 00013A A072                    ENAB                      enable SYNC interrupt *
 *  92 00013C 43FF00SCRSEM            LEA    .SCRSEM,A1         get address of sema *
 *  93 00013F 4E47                    RELEA                     release the semaphor *
 *  94 000141                 *       enable the MAUS interrupt for next movement *
 *  95 000141 203AFF1E        MEXIT:  MOVE.L MAUS,D0            get interrupt mask *
 *  96 000145 A072                    ENAB                      enable MAUS interrupt *
 *  97 000147 4E41                    TERMI                     quick end of job   *
 *  98 00014C                 *                                               *
 *  99 00014C                 ***********************************************
 *                                    END    TASK1                            *
```

In Assembler arbeitet die MOUSE-Task noch schneller.

Wichtigste Aufgabe zu Beginn der Task ist es, den Stackpointer auf den dafür vorgesehenen Bereich im Task-Workspace zu setzen (Zeile 45). Gleich anschließend wird der MAUS-Interrupt gesperrt. Auch hierfür steht ein Line-A-Trap zur Verfügung, der Echtzeitprogrammierung in Assembler komfortabel macht. Für die RELEASE- und REQUEST-Operationen gibt es ebenfalls solche Traps (Zeilen 59, 60 und 91, 92), denen nur die Adresse der Semaphore in A1 übergeben werden muß. Genau wie ein PEARL-Programm wird auch hier das Assembler-Programm in den Zustand 'waiting for semaphore' gebracht, wenn die Semaphore schon anderweitig 'requested' worden ist.

Beendet wird die Task auch wieder durch einen Trap, nämlich den Trap 1, der die Task in den Zustand 'dormant' beziehungsweise hier in den Zustand 'scheduled' überführt. Wäre die Task nicht 'resident', würde mit TERMI auch der Task-Workspace zurückgegeben. Die Kommentare und eventuell das PEARL-Programm aus Teil 3 dürften den Rest des Programms verständlich machen. Auf der PEARL-Seite sind nur geringe Modifikationen vorzunehmen. MOUSE ist jetzt als externe Task zu spezifizieren, während einige Variable sowie SCRSEM als 'GLOBAL' zu deklarieren sind.

Was bringt es, daß man sich der Mühe unterzieht, die Task in Assembler zu programmieren? Außer Frage steht, daß sich der Programmierer mehr Aufwand und eine größere Fehleranfälligkeit einhandelt.

Nun, zum einen ist gerade diese Task natürlich ein schönes Demonstrationsobjekt für die Betriebssystem-Traps. Es wird aber auch wohl schon deutlich,

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /*   M A U S   u n t e r   R T O S                               */
 4 /*   last update:  11.09.86    06h20                             */
 5 /*                                                               */
 6 /*   version for external MOUSE task                             */
 7 /*                                                               */
 8 /*****************************************************************/
 9 S=972;                          /* to spare memory   */
10 MODULE MOUSE;
11
12 SYSTEM;
13    A1:   A1 <->;                /* Terminal          */
14    SYNC: EV(40000000);          /* V-Sync. Interrupt */
15    MAUS: EV(80000000);          /* Mausinterrrupt    */
16
17 PROBLEM;
18    SPC A1      DATION INOUT ALPHIC CONTROL(ALL);
19    SPC SYNC    INTERRUPT;
20    SPC MAUS    INTERRUPT;
21    SPC (CLEAR,SCRSET,SCROHG,SCRILL,WIDTH) ENTRY GLOBAL;
22    SPC (SCLOAD,SCSAVE,SPRITL,SPRITS)      ENTRY GLOBAL;
23    SPC MOUSE   TASK GLOBAL;
24
25    DCL TRUE  INV BIT(1) INIT('1'B);
26    DCL FALSE INV BIT(1) INIT('0'B);
27
28    DCL (XWIDTH,YWIDTH) FIXED; /* screen resolution   */
29    DCL (XLIMIT,YLIMIT) FIXED; /* screen border       */
30    DCL (MX,MY) FIXED  GLOBAL; /* mouse position      */
31    DCL (MXO,MYO) FIXED;       /* old mouse position  */
32    DCL ARROWS (2,11) BIT(16); /* to save picture under arrow */
33    DCL MNAME  CHAR(5);        /* name of active picture */
34    DCL LBTN BIT(1)    GLOBAL; /* left button flag    */
35    DCL RBTN BIT(1)    GLOBAL; /* right button flag   */
36    DCL GOTN BIT(1) INIT('0'B);/* picture name flag   */
37    DCL SCRSEM SEMA GLOBAL PRESET(1);
38                               /* mouse position update semaphore */
39
40
```

Im Malprogramm aus c't 10/86 brauchen nur der Modulkopf geändert und die Task MOUSE gestrichen zu werden.

RTOS— Pe(a)rlen

In den letzten Folgen ging es um die Programmierung unter dem Betriebssystem RTOS/UH. Doch die Programmierung ist nur ein Schritt bei der Programmentwicklung, ebenso umfangreich ist danach oft das 'Debugging', das Suchen nach Fehlern, das Austesten der Programmfunktionen. Einige der Hilfsmittel, die dem Programmierer von RTOS hierfür zur Verfügung gestellt werden, möchte ich in dieser Folge vorstellen. Und ich meine, daß für diese der Untertitel durchaus gerechtfertigt ist.

daß gerade hier die Abfragen und Tests einfacher auszuführen sind als in PEARL, da der Datentyp Bit beziehungsweise Byte in Assembler einfacher zu handhaben ist. Ein weiteres Argument könnte eventuell der geringere Speicherbedarf sein; die Assembler-Task benötigt nur 60% des Platzes der PEARL-Task. Der wichtigste Pluspunkt zugunsten der Assembler-Programmierung besteht aber in der höheren Ausführungsgeschwindigkeit. Da jedoch die Rechenleistung der Computer immer weiter steigt, verliert auch dieses Argument wohl mit der Zeit an Boden.

Allerdings soll dieser Beitrag nicht ausschließlich auf die Fehlersuche beschränkt bleiben, sondern allgemein Aspekte der Programmentwicklung behandeln. Deshalb werden alle fünf Compiler-Optionen näher beschrieben.

Schon in Folge 1 dieser Reihe hatte ich erwähnt, daß nicht unbedingt alles, was zwischen '/*' und '/*' steht, von dem Compiler als Kommentar ignoriert wird. Die Compiler-Optionen lassen sich an- und abschalten, indem man als erstes Zeichen in dem Kommentar ein '+' oder '–' setzt, gefolgt von L, P, M, T oder B als Kennung der Option. Alle Optionen bis auf eine habe ich in einem kleinen Demon-

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /* D E M O N S T R A T I O N   C O M P I L E R O P T I O N E N */
 4 /* last update: 03.10.86   22h10                                 */
 5 /*                                                               */
 6 /*****************************************************************cm*/
 7 MODULE M;
 8
 9 ;/*-L  Protokoll ausschalten */;
10 SYSTEM;
11    A1;
12 PROBLEM;
13    SPC A1 DATION INOUT ALPHIC CONTROL(ALL);
14 ;/*+L  Protokoll einschalten */;
15
16 /*+P  generierten Code anzeigen */;
17 TEST:TASK;
18    DCL FIELD(8,5) FIXED;
19    DCL (X1,X2)    FLOAT(55);
20
21 /*+T  Feldgrenzenueberwachung einschalten */;
22    FIELD(1,3) = 0;
23    FIELD(9,2) = 13;
24    PUT 'FELD (1,3): ',FIELD(1,3) TO A1
25      BY SKIP,A,F(5),SKIP;
26 /*+M  Zeilenmarkierung einschalten */;
27    FIELD(6,1) = 0;
28    FIELD(5,6) = 13;
29 /*-M  Zeilenmarkierung ausschalten */;
30 /*-T  Feldgrenzenueberwachung ausschalten */;
31    X1 = 3.141592;
32    X2 = X1 * 0.815;
33    X2 = SIN(X2);
34 /*-P  generierten Code nicht anzeigen */;
35 END;
36
37 MODEND;
38
```

In diesem Quelltext werden vier Optionen des UH-Compilers ausgenutzt.

strationsprogramm untergebracht. Auf die Option 'B' werde ich später noch eingehen.

Wie bei den meisten Compilern und Assemblern, so läßt sich auch bei dem PEARL-Compiler die Ausgabe eines Listings steuern. Zum einen kann dieses beim Aufruf des Compilers geschehen, indem als LO-(List-output-)Parameter 'NO' (kein Listing) oder eine Datenstation angegeben wird. Zum andern kann mit '+L' (Zeile 8) beziehungsweise '-L' (Zeile 3) im Programm die Ausgabe des Listings gestartet und wieder unterbrochen werden. Dabei wird nach '+L' auch dann ein Listing ausgegeben, wenn beim Compiler-Aufruf 'LO NO' angegeben wurde. Das Semikolon zu Beginn der Zeilen 3 und 8 habe ich gesetzt, weil die List-Option immer von dem letzten vorangehenden Semikolon an wirkt. Stände etwa in Zeile 8 kein Semikolon vor dem Kommentar, so würde auch Zeile 7 aufgelistet.

Compiler-Code sichtbar gemacht

Noch aus dem Stadium der Compiler-Entwicklung stammt die P-Option, mit der der vom Compiler generierte Code angezeigt werden kann. Ab Zeile 10 ist deshalb vor jeder Programmzeile zu sehen, was der Compiler daraus gemacht hat. Wer allerdings hofft, ein Assemblerlisting zu bekommen wie bei einigen C-Compilern, das so oder modifiziert dem Assembler angeboten werden kann, wird enttäuscht sein. Das Listing des Demonstrationsprogrammes sieht denn auch auf den ersten Blick überhaupt nicht nach 68000-Code aus.

Zum einen liegt das daran, daß der Compiler für komplexere Funktionen wie Ein-/Ausgabe auf den in Teil 4 beschriebenen Hyperprocessor zurückgreift. Dies kann man daran erkennen, daß in dem Code-Listing '-TO-VIRTUAL' auftaucht. Mit '-TO-REAL' wird analog wieder auf 68000-Code zurückgeschaltet. Zum andern werden andere Mnemonics verwendet. So verbirgt sich hinter D0*2 ein ADD.L D0,D0 oder hinter D0A0 ein MOVEA. L D0,A0. Es können sich sogar mehrere Instruktionen hinter einem Mnemonic verbergen, wie etwa bei MOVD. Das D steht dabei für 'double' und bedeutet, daß eine FLOAT(55)-Zahl bewegt werden soll. Das ist natürlich mit einer Instruktion nicht mehr zu machen.

Wer etwas mehr an dem vom Compiler generierten Code interessiert ist, findet eine knappe Beschreibung der Compiler-Mnemonics im Handbuch, Abschnitt E-IV. Der Assembler-Programmierer, der die Betriebssystem-Traps oder den Hyperprocessor verwenden möchte, sollte sich kleine Programme schreiben, in denen nur die Befehle enthalten sind, die genauer untersucht werden sollen, und die Listings mit der P-Option ausdrucken lassen. Nach dem Laden kann man dann mit dem S-Befehl herausfinden, an welchem Speicherplatz das Programm steht, und

es mit dem Monitor disassemblieren. Vereinfachen kann man sich das Rechnen mit Adressen (die P-Option zeigt nur Adressen relativ zum Modulanfang an), indem man das Programm auf eine gerade Adresse lädt. Dafür gibt es beim LOAD-Befehl den Parameter AD. Mit

LOAD AD 10000

wird das Programm ab Adresse sedezimal 10000 geladen (wenn es nach ED.SR übersetzt worden ist). Man muß nur vorher kontrollieren, ob dort auch tatsächlich Platz ist, sonst meldet der Lader

```
> > LOADxx:
  MEMORY OVERFLOW
  LOADER INPUT
```

und bricht den Ladevorgang ab.

Noch eine Warnung an alle Assembler-Programmierer, die den Hyperprocessor benutzen wollen: Es ist zu erwarten, daß der Hyperprocessor im Zuge der Weiterentwicklung des Compilers ohne besondere Ankündigung geändert wird. So werden die Hyperprocessor-Befehle, die in dem Handbuch mit dem Hinweis 'Nicht für Neuentwicklung' stehen, bereits von dem Compiler D9.9 nicht mehr benutzt. Wenn sie dennoch im Hyperprocessor verblieben sind, dann nur, damit ältere Übersetzungen von Programmen noch laufen und weil die Codes noch nicht wieder benötigt wurden. Es empfiehlt sich deshalb immer, PEARL-Programme auf dem Rechner zu übersetzen, auf dem sie ausgeführt werden sollen, sowie nach einem Compiler-/Hyperprocessor-Update alle Programme neu zu übersetzen. Bei Assembler-Programmem sollte beim Übertragen auf andere PEARL-Rechner überprüft werden, ob der dort installierte Hyperprocessor die verwendeten Funktionen auch enthält.

Doch nicht allein für den Assemblerprogrammierer ist die P-Option interessant. Tritt in einem Programm ein Fehler auf, der das Betriebssystem veranlaßt, das Programm abzubrechen, so wird eine Fehlermeldung ausgeben. Diese Fehlermeldung enthält neben dem Programmnamen und einer Kurzbeschreibung des Fehlers auch den Stand des Program Counters, bei dem das Programm abgebrochen wurde.

Diese Adresse sollte man sich notieren, denn wenn der Fehler nicht zufällig in einer Betriebssystemroutine aufgetreten ist (das heißt, im Adreßbereich FA0000 . . . FBFFFF), läßt sich damit der Fehler meist schon so ungefähr lokalisieren. Dazu muß die Anfangsadresse des Modules, das die fehlerhafte Task enthält, von der Adresse in der Fehlermeldung abgezogen werden. Mit dieser relativen Adresse geht man nun in das mit eingeschalteter P-Option übersetzte Programm und findet damit den Programmteil, in dem der Fehler auftrat. Natürlich kann dann die Fehlerursache noch ganz woanders liegen, aber einen Anhaltspunkt hat man auf diese Weise schon mal.

Wesentlich einfacher gestaltet sich die Fehlersuche, wenn das Programm mit der M-Option übersetzt wurde. M steht für mark oder Markierung und bedeutet, daß das Programm die Nummer der Programmzeile, die es gerade abarbeitet, an einer bestimmten Stelle speichert. Tritt nun wie oben ein Fehler auf, so wird nicht der Stand des Program Counters, sondern die zuletzt gespeicherte PEARL-Zeilennummer in der Fehlermeldung ausgegeben. Diese muß aber nicht unbedingt die Zeilennummer der zuletzt abgearbeiteten Zeile sein, da die Markierungsoption ja auch abschaltbar ist und dann nicht mehr die aktuelle Zeilennummer gespeichert wird.

Wie man in dem Compiler-Code der Zeilen 17 und 18 erkennen kann, wird zur Speicherung der Zeilennummer der Line-A-Trap A036 mit dem Mnemonic LIMR verwendet, hinter dem die Zeilennummer steht. Gespeichert wird die Zeilennummer im Task-Workspace bei der relativen Adresse $00A2. Mit dem Kommando DL des Betriebssystems läßt sich diese Nummer anzeigen. Man kann sich diese Nummer auch während des Laufs eines Programmes anzeigen lassen und so eventuell grob abschätzen, wie weit ein Programm gerade ist. Eine Möglichkeit zur Fehlersuche besteht darin, in verschiedenen Programmzweigen ein 'SUSPEND' einzubauen. Mit DL läßt sich dann feststellen, in welchem Programmzweig das Programm abgebrochen wurde.

Eine beliebte und verbreitete Fehlerursache ist die Überschreitung von Feldgrenzen. Dieser Fehler ist besonders angenehm, da er sich häufig erst später auswirkt. Dann kann aber zum Beispiel die Verzeigerung der Speicherblöcke schon zerstört sein, was das Betriebssystem mit einem Kaltstart honoriert. Daß aber selbst mit der Meldung 'RESET' auf dem Bildschirm noch nicht alles verloren ist, zeige ich später noch.

Gegen die Index-Fehler ist im UH-PEARL-Compiler ein Kraut namens T-Option gewachsen. Diese überprüft (natürlich nur, wenn eingeschaltet) während der Programmausführung, ob mit den gegebenen Indizes die Feldgrenze nicht überschritten wird. Bei eindimensionalen Feldern und Zeichenketten (die T-Option wirkt auch für

```
=    1 /*!!!
```

halb der Felder gegen Überschreiben gewährleistet ist. Innerhalb eines Feldes kann aber ein Index durchaus trotzdem den gültigen Bereich, der in der Deklaration des Feldes festgelegt ist, verlassen.

In dem Demonstrationsprogramm ist dies in den Zeilen 22 bis 28 zu sehen. In Zeile 18 ist FIELD als 8 x 5 großes Feld von FIXED-Werten vereinbart. In Zeile 22 und 23 weise ich nun zwei Feldelementen Werte zu, wobei bei der zweiten Zuweisung eine Feldgrenzenüberschreitung erfolgt. Dasselbe passiert in den Zeilen 27 und 28, nur die Feldgrenzenüberschreitung passiert in der anderen Dimension. Läßt man das übersetzte Programm nun laufen, so erscheint

```
*TEST
FELD (1,3): 13
>> TESTI=0028 WRONG
   INDEX
```

auf dem Bildschirm. Offensichtlich ist die erste Überschreitung nicht entdeckt worden. Der ausgegebene Wert '13' läßt aber schon ahnen, woran dies liegt: die Feldelemente (x,3) sind direkt im Anschluß an die Feldelemente (x,2) gespeichert. So wird mit Zeile 23 das Feld (1,3) überschrieben, da sowohl 2 x 8 + 9 = 25 als auch 3 x 8 + 1 = 25 auf denselben eindimensionalen Index führen. Da 25 aber kleiner oder gleich 5 x 8 + 8 = 48 ist, wird kein Fehler entdeckt. Anders im zweiten Fall. Dort wird in Zeile 28 durch 6 x 8 + 5 = 53 der Speicherbereich des Feldes verlassen, was mit der T-Option auch prompt als Fehler erkannt wird.

Sowohl die M- als auch die T-Option veranlassen den Compiler, mehr Code zu erzeugen (der Line-A-Trap ITS2 mit den beiden Zeilen vorher ist nur für die Feldgrenzenüberprüfung). Da dies sich natürlich in einem höheren Zeitverbrauch niederschlägt, werden diese Optionen zumeist aus den fertigen Programmen entfernt und nur beim Auftauchen von Fehlern wieder eingebaut.

Als fünfte Option gibt es noch die oben erwähnte B-Option, wobei das B für 'big module' steht. Durch diese Option wird der Compiler veranlaßt, nach jeder Prozedur oder Task den Konstanten-Pool zu leeren. Der Compiler speichert längere Zahlenkonstanten und Zeichenketten nicht im Programmcode, sondern sammelt sie und legt sie normalerweise am Ende eines Modules ab. Aus dem Programm wird dann mit PC-relativer Adressierung auf diese Konstanten zugegriffen. Da dabei der Ort im Programm, von dem aus zugegriffen wird, und der Ablageplatz der Konstante nicht weiter als 32000 auseinanderliegen dürfen, bedeutet dieses Verfahren eine Beschränkung der Modulgröße auf 32 KByte. Diese läßt sich mit der B-Option wieder aufheben: in langen Programmen mit vielen Konstanten kann dadurch der Compiler veranlaßt werden, nach jeder Prozedur oder Task einen Ablageplatz für Konstanten einzurichten. Dadurch darf jede Task und Prozedur bis zu 32 KByte lang werden, eine Obergrenze, die in der Praxis wohl kaum jemals in Erscheinung treten wird.

Tracing auf PEARL-Ebene

Nach den Möglichkeiten, die die Compiler-Optionen bei der Fehlersuche bieten, sollen jetzt noch einige Befehle des Kommando-Interpreters vorgestellt werden. Da ist zunächst der Befehl

TRACE taskname L lineno
(,lineno)

wobei lineno eine 4stellige Zahl ist. Damit dieser Befehl etwas bewirkt, muß das Programm mit eingeschalteter M-Option übersetzt worden sein.

Durch den TRACE-Befehl wird die betreffende Task jedesmal vor Bearbeitung der angegebenen Zeile suspendiert. Da der Linemarker-Trap, wie in dem Demonstrationsprogramm für die Compiler-Optionen zu sehen, jeweils als erste Instruktion vor dem restlichen Code einer Programmzeile steht, ist diese Programmzeile zum Zeitpunkt der Unterbrechung noch nicht ausgeführt. Durch eine Meldung erfährt man, daß eine eingegebene Zeilennummer erreicht wurde und welche es war, wenn zwei angegeben waren. Jetzt kann man zum Beispiel eine Task starten, die Variablen anzeigt. Durch einen erneuten TRACE-Befehl kann man dann die Zeilennummer(n) ändern, bei der/denen unterbrochen werden soll. Die unterbrochene Task kann mit

CONTINUE taskname

wieder gestartet werden. Wurde die Zeilennummer vor dem CONTINUE nicht geändert, so bleibt sie bestehen. Man kann also Schleifen bei einmaliger Eingabe einer Zeilennummer ohne neues TRACE-Kommando jeweils eine Runde laufen lassen. Die Zeilenüberwachung wird erst durch den Befehl

NOTRACE taskname

aufgehoben.

. . . und in Assembler

Aber nicht nur für PEARL-

Programme läßt sich der TRACE-Befehl einsetzen. Man kann auch statt 'L lineno' eine sechsstellige Sedezimaladresse (allerdings nur eine) angeben und damit Assemblerprogramme an der angegebenen Adresse anhalten. Allerdings gibt es einige Bedingungen, wie etwa absolute Sprünge, durch die die Programmzähler-Überwachung lahmgelegt wird. Mit dem TRACE-Befehl wird nämlich Bit 31 des Programmzählers gesetzt, so daß auch mit BSR aufgerufene Prozeduren verfolgt werden können. Wird allerdings der Programmzähler neu geladen, so wird Bit 31 zurückgesetzt; die angegebene Adresse kann dann nicht mehr erkannt werden, da auch auf das gesetzte Bit 31 geprüft wird.

Hat man eine Assembler-Task auf diese Weise angehalten, kann man sich die augenblicklichen Registerinhalte mit dem Befehl

DR taskname

anzeigen lassen. Jedesmal, wenn eine Task unterbrochen wird, muß der Zustand der Task gespeichert werden, damit sie anschließend ohne Schwierigkeiten weiterlaufen kann. Zu den Informationen, die den Zustand einer Task ausmachen, gehören auch die Registerinhalte. Deshalb ist im Task-Workspace jeder Task Speicherplatz für sie reserviert, der mit dem DR-Befehl angezeigt werden kann. Das Anzeigeformat ist aus Einfachheitsgründen das des DM-Befehls (display memory) – DR tut ja letztlich dasselbe – und von daher nicht sehr übersichtlich. Welcher Wert zu welchem Register gehört, läßt sich dieser Tabelle entnehmen:

Adr.:	A7(USP)	A7(SSP)	D0	D1
Adr.:	D2	D3	D4	D5
Adr.:	D6	D7	A0	A1
Adr.:	A2	A3	A4	A5
Adr.:	A6	St 5/4	St 3/2	St 1/0

In diesem Format werden beim DR-Befehl die Register angezeigt.

Die mit St gekennzeichneten Stellen enthalten die letzten drei Langwörter des Systemstacks, Adr. gibt die Speicherstelle an, an der die Register gespeichert sind. Um Registerinhalte zu ändern, braucht man nur mit dem SM-Befehl (set memory) die neuen Werte an die entsprechenden Speicherstellen zu schreiben. Läuft die Task dann weiter, etwa nach einem CONTINUE, so werden die geänderten Werte in die Register übernommen.

Retten, was zu retten ist

Eigentlich würde man Befehle wie DM (display memory) und SM (set memory) eher bei einem Monitor als bei einem Kommandoprozessor erwarten. Um so mehr freut man sich natürlich, denn sie lassen Eingriffe zu, die von anderen Betriebssystemen nicht so einfach gestattet werden. Als Beispiel dafür möchte ich demonstrieren, wie nach einem Rechnerabsturz und dem damit verbundenen Kaltstart trotzdem noch Files im Speicher gerettet werden können. Bild 1 zeigt die Speicherbelegung vor dem Absturz, Bild 2 die danach. Wie zu sehen, sind alle Files und Module gelöscht, der Bereich zwischen 262A und F73F2 ist mit FREE angegeben. Doch der Schein trügt, kein einziges Byte ist gelöscht, nur die Verzeigerung ist neu aufgesetzt worden.

RTOS verwaltet den Speicher, indem es ihn in Speicherblöcke aufteilt. Am Anfang jedes Speicherblockes stehen zwei Adressen, nämlich die Adresse des sich jeweils direkt vorne und hinten anschließenden Speicherblockes. Wenn man die Adresse nur eines Speicherblockes weiß, kann man also die Adressen aller Speicherblöcke herausfinden. So ein Verweis auf den nächsten Datenblock wird auch als Link bezeichnet; da jeweils zwei vorhanden sind, unterscheidet man zwischen FORL (forward link, Vorwärtsverweis) und BACKL (backward link, Rückwärtsverweis).

Mit dem Kaltstart wird der FORL der Anfangsmarke auf die Endmarke des Speichers gesetzt und der BACKL der Endmarke auf die Anfangsmarke. Außerdem wird der so entstandene Speicherblock mit dem Typ 'FREE' gekennzeichnet. Wenn man die Verzeigerung der Speicherblöcke, die vor dem Absturz im Rechner waren, wiederherstellen kann, sind häufig die Dateien noch zu retten.

Doch dazu muß man einen Ansatzpunkt finden: Ich wußte noch, daß in dem ersten Speicherblock der Compiler lag und in dem zweiten der Assembler. Also machte ich mich mit dem DM-Befehl auf die Suche nach dem Namen des zweiten Modules, 'ASSEM'. Da der Name ab Byte 10 in einem Modul steht, war der Modulanfang schnell

Speicherblock:	FORL	Vorwärts-Verweis
	BACKL	Rückwärts-Verweis
	TYPE	Typ des Speicherblockes ($10 = MDLE)
bei MDLE	NAME	6 Buchstaben Name

gefunden. Wie erwartet, zeigte BACKL noch auf den ersten Speicherblock. Jetzt mußte nur noch FORL des ersten Speicherblockes auf das Modul 'ASSEM' gerichtet werden, und schon zeigte sich mit dem S-Kommando der Speicher wieder gefüllt. Allerdings erschien der Block mit dem Compiler noch als FREE. Durch Ändern des Type-Bytes war aber auch das schnell korrigiert.

Das S-Kommando braucht nur FORL, um alle Speicherblöcke anzuzeigen. Für die Routinen, die den BACKL benutzen (etwa beim Suchen nach Task Workspace), ist noch der BACKL der Endmarke auf den vorletzten Block zu setzen. Damit war die Verzeigerung der Speicherverwaltung schon restauriert. Nur der ED-Filemanager wußte noch nichts von dem Glück, daß ED.EDFILE gerettet war. Das liegt daran, daß die Speicherblöcke, die der ED-Filemanager verwaltet, durch eine Extra-Verweiskette verbunden sind. Um ED.EDFILE dort wieder anzuhängen, eröffnete ich einfach ein File ED.SI. Bei Offset $20 stand dort 0000 0000 als Zeichen dafür, daß es keine weiteren ED-Files gibt. Dort war jetzt nur noch die Adresse von ED.EDFILE einzutragen, und auch der ED-Filemanager kannte EDFILE wieder.

Ein wenig aufpassen muß man natürlich, wenn der Kaltstart aus einem Programmfehler herrührt. Ich war zu meinem mißlichen Kaltstart gekommen, weil ich versehentlich den Monitor mit 'RT' statt mit 'RW' verlassen hatte. Da ich im Monitor keine Speicherveränderungen vorgenommen hatte, konnte ich ziemlich sicher sein, daß bis auf die oben beschriebenen Links die übrige Verzeigerung keinen Schaden genom-

Rekonstruktion der Speicherorganisation nach einem Reset

```
#S
0022AB->0022B2  MARK
0022B2->0022FC  ATSK  Resident  #IDLE
0022FC->002346  TASK  Resident  #ACIA1
002346->002390  TASK  Resident  #RS232
002390->0023DA  TASK  Resident  #MIDIO
0023DA->002424  TASK  Resident  #PPORT
002424->00246E  TASK  Resident  #EDFMN
00246E->0024BB  ATSK  Resident  #ERROR
0024BB->002502  TASK  Resident  #VDATN
002502->00254C  TASK  Resident  #USER0
00254C->002596  TASK  Resident  #USER1
002596->0025E0  TASK  Resident  #XCMMD
0025E0->00262A  TASK  Resident  #UHFX6
00262A->008A46  MDLE  PCOMP
008A46->00A516  MDLE  ASSEM
00A516->00A558  MDLE  PRINT
00A558->00AA16  TASK  SETFRN
00AA16->00BA16  EDTF  EDFILE
00BA16->0F73F2  FREE
0F73F2->000000  MARK
```

① *So sah die Speicherbelegung vorher aus –*

```
#S
0022AB->0022B2  MARK
0022B2->0022FC  ATSK  Resident  #IDLE
0022FC->002346  TASK  Resident  #ACIA1
002346->002390  TASK  Resident  #RS232
002390->0023DA  TASK  Resident  #MIDIO
0023DA->002424  TASK  Resident  #PPORT
002424->00246E  TASK  Resident  #EDFMN
00246E->0024BB  ATSK  Resident  #ERROR
0024BB->002502  TASK  Resident  #VDATN
002502->00254C  TASK  Resident  #USER0
00254C->002596  TASK  Resident  #USER1
002596->0025E0  TASK  Resident  #XCMMD
0025E0->00262A  TASK  Resident  #UHFX6
00262A->0F73F2  FREE
0F73F2->000000  MARK
```

② *so nach dem ungewollten Kaltstart.*

```
#DM 8A46 20
008A46: 0000 A516 0000 262A 0010 4153 5345 4D20  ......%&..ASSEM.
008A56: 8034 95CF 0000 0000 BBCE 81C4 9564 955B  .4...........d.[
008A66: 0018 801C 0014 0047 016B 0000 0202 0404  .......G.h......
```

③ Jetzt wird nach dem Namen des zweiten Moduls gesucht: 'ASSEM'.

```
#DM 262A 20
00262A: 000F 73F2 0000 25E0 0000 5043 4F4D 5020  ..s...%...PCOMP.
00263A: 99FF 9541 B09A B4C6 A201 A9F9 A9F3 0000  ...A............
00264A: 0038 8074 0018 0047 02D0 DCD8 5349 4E80  .8.t...G....SIN.
```

④ dann nach dem Namen des ersten: 'PCOMP'.

```
#SM 262A 0000 8A46
```

⑤ Der Zeiger FORL wird auf den Anfang des zweiten Moduls gerichtet.

```
#S
0022AB->0022B2  MARK
0022B2->0022FC  ATSK  Resident  #IDLE
0022FC->002346  TASK  Resident  #ACIA1
002346->002390  TASK  Resident  #RS232
002390->0023DA  TASK  Resident  #MIDI3
0023DA->002424  TASK  Resident  #PPORT
002424->00246E  TASK  Resident  #EDFMN
00246E->0024B8  ATSK  Resident  #ERROR
0024B8->002502  TASK  Resident  #VDATN
002502->00254C  TASK  Resident  #USER0
00254C->002596  TASK  Resident  #USER1
002596->0025E0  TASK  Resident  #XCMMD
0025E0->00262A  TASK  Resident  #UHFX6
00262A->00BA46  FREE
00BA46->00A516  MDLE  ASSEM
00A516->00A55B  MDLE  PRINT
00A55B->00AA16  TASK  SETPFN
00AA16->00BA16  EDTF  EDFILE
00BA16->0F73F2  FREE
0F73F2->000000  MARK
```

⑥ Noch erscheint der Compiler als 'FREE' in der Speicherbelegung.

```
#SM 262A+8 0010
```

⑦ durch Ändern des Typ-Bytes ist das schnell korrigiert.

```
#DM F73F2 20
0F73F2: 0000 0000 0000 262A 8000 0000 FFFF 0000  ......%&........
0F7402: 0400 0400 0001 001B 0000 0000 000A 010D  ................
0F7412: 001B 0000 0000 0000 0000 0000 0000 001B  ................
```

```
#SM F73F2+4 0000 BA16
```

⑧⑨ Jetzt wird BACKL der Endemarke korrigiert.

men hatte. Nach einem Kaltstart durch Programmfehler sollte man aber alle Links überprüfen und gegebenenfalls ausbessern.

Natürlich besteht die beschriebene Möglichkeit zum Restaurieren der Zeiger nur, wenn der Speicher beim Kaltstart nicht gelöscht wird. Schwierigkeiten sind bei der nächsten RTOS-UH-Version für den Atari ST zu erwarten, die zur Zeit vorbereitet wird. Und zwar aus folgendem Grund: Es hat sich herausgestellt, daß die Video-Initialisierung nach einem Kaltstart beim 1040ST nicht immer einwandfrei verläuft. Schuld daran ist eine winzige Inkompatibilität, deren Ursache bisher nicht aufzuspüren war. Deshalb wird bei der künftigen Version die Initialisierung noch dem TOS überlassen; der Aussprung in die RTOS-Welt erfolgt erst später. Dann hat aber TOS – warum nur!? – zum Teil den Speicher gelöscht, und nach einem Reset ist tatsächlich nicht mehr viel zu retten.

Ein anderes Beispiel für Manipulationsmöglichkeiten ist das Verändern von Task-Prioritäten. Es gibt Ausnahmefälle, in denen eine Task nicht solange warten darf, wie etwa die ERROR-Task zur Ausgabe einer Fehlermeldung braucht. Beim ATARI mit seiner Terminalemulation ist ja jede Zeichenausgabe recht zeitaufwendig. Solche Tasks müssen dann auf eine höhere (negative) Priorität gesetzt werden. Da RTOS aber dem Benutzer nicht gestattet, negative Zahlen als Prioritätsparameter anzugeben, muß 'von Hand' nachgeholfen werden. Aus der letzten Folge dieser Serie ist ersichtlich, daß bei Offset $10 und $1E im Task-Kopf die Priorität steht. Weiß man die Adresse einer Task, kann man also mit

```
#S
0022AB-0022B2  MARK
0022B2-0022FC  ATSK  Resident  #IDLE
0022FC-002746  TASK  Resident  #ACIA1
002746-002390  TASK  Resident  #RS232
002390-0023DA  TASK  Resident  #MIDI3
0023DA-002424  TASK  Resident  #PPORT
002424-00246E  TASK  Resident  #EDFMN
00246E-0024B8  ATSK  Resident  #ERROR
0024B8-002502  TASK  Resident  #VDATN
002502-00254C  TASK  Resident  #USER0
00254C-002596  TASK  Resident  #USER1
002596-0025E0  TASK  Resident  #XCMMD
0025E0-00262A  TASK  Resident  #UHFX6
00262A-00BA46  MDLE  PCOMP
00BA46-00A516  MDLE  ASSEM
00A516-00A558  MDLE  PRINT
00A558-00AA16  TASK  SETPRN
00AA16-00BA16  EDTF  EDFILE
00BA16-00CA16  EDTF  SI
00CA16-0F73F2  FREE
0F73F2-000000  MARK
```

❶ *Alle Files sind wieder vorhanden. Um auch die eigene Verweiskette des Editor-Filemanagers zu restaurieren, wird eine neue Editor-Datei SI eröffnet . . .*

```
#DM BA16 30
00BA16: 0000 CA16 0000 AA16 0020 0000 002A 0000  .............*..
00BA26: 1542 0000 0000 0000 0000 0000 003B 0FFE 0056  .B.........8...V
00BA36: 0000 0000 0000 BA16 003B 8040 0020 0000  .........8.@....
00BA46: 0200 0001 5349 FF49 0001 1B2A 4649 4C45  ....SI.I...*FILE

#SM BA16+20 0000,AA16
```

❷❷ *. . . und in deren Kopf der Verweis auf die Adresse der vorangehenden Editor-Datei eingetragen.*

*SM xxxx+10 FFF0

*SM xxxx+1E FFF0

einfach die Priorität auf einen negativen Wert (hier -10h) setzen und damit der Task höchste Priorität verleihen.

Simulation von Interrupts

Zum Schluß möchte ich noch ein Bonbon vorstellen, das RTOS-UH für den Programmentwickler bietet. Mit dem Befehl

TRIGGER EV interruptmask

lassen sich Hardware-Interrupts simulieren; dabei werden die zugehörigen interruptgesteuerten Programme aktiviert. Die Programme können nicht zwischen 'echten' Hardware-Interrupts und diesen simulierten unterscheiden. Durch Setzen der korrespondierenden Bits der Interruptmaske (siehe Teil 5 dieser Serie) kann bestimmt werden, welche Interrupts ausgelöst werden sollen. TRIGGER EV C0000000 würde beim ATARI also Maus- und Bildrücklaufinterrupt simulieren. In meinem kleinen Simulationsprogramm für die ATARI-Maus wird die Bytefolge, wie sie von der Maus käme, in den dafür vorgesehenen Buffer geschrieben und dann über den Kommando-Kanal XC. die TRIGGER-Anweisung ausgegeben. Zur Erinnerung: Zeichenketten, die in den Kommando-Kanal XC. geschrieben werden, haben dieselbe Wirkung, als würden sie über das Bedien-Interface eingegeben. Mit

ALL 1 SEC RIGHT

kann dann nach dem Laden eine gleichmäßige Mausbewegung

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /*  M A U S   S I M U L A T I O N                                */
 4 /*  last update:  04.10.86    08h15                              */
 5 /*                                                               */
 6 /*****************************************************************/cem*/
 7 S=2B0;                          /* to spare memory    */
 8 MODULE SIMULA;
 9
10 SYSTEM;
11    XC:   XC ->;                 /* Kommando Kanal     */
12    MAUS: BU(03000BE6) ->;       /* Mausdaten          */
13
14 PROBLEM;
15    SPC XC     DATION   OUT ALPHIC CONTROL(ALL);
16    SPC MAUS   DATION   OUT BASIC;
17
18 /*****************************************************************/
19 /* simulate mouse movement                                       */
20 /*****************************************************************/
21 RIGHT:TASK;
22    SEND 'FB010000'B4 TO MAUS;
23    PUT 'TRIGGER EV B0000000' TO XC;
24 END;
25
26 LEFT:TASK;
27    SEND 'FBFF0000'B4 TO MAUS;
28    PUT 'TRIGGER EV B0000000' TO XC;
29 END;
30
31 UP:TASK;
32    SEND 'FB00FF00'B4 TO MAUS;
33    PUT 'TRIGGER EV B0000000' TO XC;
34 END;
```

```
33
36 DOWN;TASK;
37    SEND 'FB000100'B4 TO MAUS;
38    PUT 'TRIGGER EV 80000000' TO IC;
39 END;
40
41 MODEND;
```

Ein Programm simuliert die Maus. Der Hardware-Interrupt kann ebenfalls nachgebildet werden – sehr nützlich in der Erprobungsphase. ct

nach rechts simuliert werden. Bewegt man zusätzlich die Maus, so mischen sich 'echte' und 'unechte' Interrupts. Die Simulation des Bildschirmrücklauf-Interrupts ist von daher etwas schwieriger, da mit DISABLE EV 40000000 auch der simulierte Interrupt unterbunden wird. Hier kann man sich helfen, indem man den echten Interrupt enabled, und die Interrupt-Task ihn als erstes wieder disabled. Auf diese Weise wird gezielt ein einzelner Interrupt ausgelöst.

Wir machen Musik

Wenn auch das Musikmachen nicht zu den typischen Aufgaben eines Prozeßrechners gehört, so bietet ein Echtzeitbetriebssytem dennoch auch dafür gute Voraussetzungen. Es soll deshalb demonstriert werden, wie man unter RTOS den Soundchip des Atari ST ansprechen kann. Dazu sind Hilfsroutinen in Assembler notwendig, die in diesem Beitrag vorgestellt werden. Danach biete ich denjenigen ST-Besitzern, die Dateien aus der TOS-in die RTOS-Welt übertragen wollen, einen Notbehelf an, der verwendet werden kann, solange unter RTOS das TOS-Diskettenformat noch nicht zur Verfügung steht.

Sehr einfach hätte man den Soundchip des Atari ST unter PEARL als Datenstation vom Typ BASIC ansprechen können. Doch leider haben die Entwickler dafür gesorgt, daß der Zugriff nur im Supervisor-Modus des Prozessors stattfinden kann. Man wollte wohl die unbefugte Benutzung verhindern, weil der Soundchip nicht nur für die Musik, sondern mit seinen beiden 8-Bit-Ports auch für die Steuerung des Diskettenlaufwerks, den Druckerausgang sowie die RS-232-Schnittstelle zuständig ist.

Wenn man dennoch unter PEARL mit dem Soundchip Musik machen will, sind Assemblerroutinen notwendig. An ihrem Anfang steht der bereits aus Teil 4 bekannte Modulkopf (der Name SNDSBR ist aus SouNDSuBRoutines entstanden). Es folgen die notwendigen Definitionen für die Benutzung des Hyperprozessors und der Betriebssystem-Traps. Den Prozeduren, die Workspace benötigen, ist die jeweilige Workspace-Definition direkt vorangestellt.

FREQA dient nun dazu, die Frequenz und damit die Tonhöhe am Analogausgang A des Soundchips einzustellen. Dazu wird als Parameter ein FIXED(15)-Wert übergeben, von dem allerdings nur die unteren 12 Bit die Frequenz bestimmen. De facto bezieht sich der übergebene Wert auf die Periodendauer des Signals: Je kleiner er ist, desto höher der erzeugte Ton. Nach der Workspace-Anforderung durch ENTR wird mit INVW der Parameter übergeben. Mit EPAR wird schließlich die Parameterübergabe abgeschlossen und der Hyperprozessor ausgeschaltet.

OFF steht als Mnemonic für den Trap 15, mit dem das Betriebssystem aufgefordert wird, alle Interrupts zu unterbinden und in den Supervisor-Modus zu wechseln. Dies ist aufgrund der Architektur des Soundchips unbedingt notwendig. Seine 16 Register werden nämlich über nur zwei Adressen angesprochen. Dazu schreibt man auf die eine Adresse die gewünschte Registernummer und auf die andere die zugehörigen Daten (Zeile 33/34 und 35/36). Soll ein Register gelesen werden, so erfolgt dies stets über ein und dieselbe Adresse (Zeile 59/60). Bei dieser Zugriffsmethode kann man aber das Pech haben, daß ein Interrupt dazwischenkommt und daß in der Interruptroutine eine andere Registernummer eingeschrieben wird. Das Kommando OFF sorgt dafür, daß diesbezüglich nichts mehr zu befürchten ist.
Nach dem Einschreiben der Frequenz wird dann in Zeile 37 das Supervisorbit im Statusregister zurückgesetzt, und die Prozedur gibt mit RETN ihren

```
 1 000000                 ****************************************************
 2 000000                 *                                                  *
 3 000000                 *   Subroutines for sound chip access              *
 4 000000                 *   last update:  22.10.86     18h40               *
 5 000000                 *                                                  *
 6 000000                 ****************************************************CRW
 7 000000 00000000              DC.L    0                   for loader        *
 8 000004 00000000              DC.L    0                                     *
 9 000008 0010                  DC      $10                 type: module      *
10 00000A 534E44534252          DC.B    'SNDSBR'            module name       *
11 000010                 ****************************************************
12 000010                 *      hyperprocessor and trap definitions         *
13 000010                 INVW   OFD.V  14                  xfer FIXED(15) by val *
14 000010                 EPAR   OFD.V  19                  end of parameter xfer *
15 000010                 ENTR   OFD.V  29                  procedure entry   *
16 000010                 RETN   OFD    $4E75               return from procedure *
17 000010                 OFF    OFD    $4E4F               switch dispatcher off *
18 000010                 ****************************************************
19 000010                 *      address definitions                         *
20 000010 FFFFB800        PSGA   EQU    $FFFFB800           sound chip reg select *
21 000010 FFFFB802        PSGD   EQU    $FFFFB802           sound chip write data *
22 000010                 ****************************************************
23 000010                 *      set frequency of analog output A            *
24 000010                 *--------------------------------------------------*
25 000010                 *      procedure workspace definitions             *
26 000010 00000000        PARO   EQU    0                   for parameter of FREQA *
27 000010 00000002        WSZO   EQU    PARO+2              size of workspace *
28 000010                 ****************************************************
29 000010 1DC000000002    FREQA  ENTR   WSZO.L              get workspace     *
30 00001a 0E000000              INVW   PARO.X               get par by value  *
31 00001A 1300                  EPAR                        end of parameter xfer *
32 00001C 4E4F                  OFF                         to supervisor mode *
33 00001E 13FC0000FFFF
        B800                    MOVE.B #0,PSGA              select register 0 *
34 00002a 13ED0001FFFF
        B802                    MOVE.B PARO+1.X,PSGD        write low byte    *
35 00002E 13FC0001FFFF
        B800                    MOVE.B #1,PSGA              select register 1 *
36 00003A 13ED0000FFFF
        B802                    MOVE.B PARO.X,PSGD          write high byte   *
37 00003E 027CDBFF              ANDI   #$DBFF,SR            allow interrupts  *
38 000042 4E4C                  RETN                        return to caller  *
39 000044                 ****************************************************
40 000044                 *      set volume of analog output A               *
41 000044                 *--------------------------------------------------*
42 000044                 *      procedure workspace definitions             *
43 000044 00000000        PAR1   EQU    0                   for parameter of VOLA *
44 000044 00000002        WSZ1   EQU    PAR1+2              size of workspace *
45 000044                 ****************************************************
46 000044 1DC000000002    VOLA   ENTR   WSZ1.L              get workspace     *
47 00004A 0E000000              INVW   PAR1.X               get par by value  *
48 00004E 1300                  EPAR                        end of parameter xfer *
49 000050 4E4F                  OFF                         to supervisor mode *
50 000052 13FC0008FFFF
        B800                    MOVE.B #8,PSGA              select register 8 *
51 00005A 13ED0001FFFF
        B802                    MOVE.B PAR1+1.X,PSGD        write volume data *
52 000062 027CDBFF              ANDI   #$DBFF,SR            allow interrupts  *
53 000066 4E4C                  RETN                        return to caller  *
54 00006B                 ****************************************************
55 00006B                 *      switch analog output A on                   *
56 00006B                 ****************************************************
57 00006B 0000            AON    VO                          hyperprocessor off *
58 00006A 4E4F                  OFF                         to supervisor mode *
59 00006C 13FC0007FFFF
        B800                    MOVE.B #7,PSGA              select register 7 *
60 000074 103FFFFFB800          MOVE.B PSGA,D0              get control register *
61 00007A 08B00000              BCLR   #0,D0                switch A on       *
62 00007E 13C0FFFFB802          MOVE.B D0,PSGD              save control register *
63 000084 027CDBFF              ANDI   #$DBFF,SR            allow interrupts  *
64 000088 4EEB0002              JMP    2(A0)                return to caller  *
65 00008C                 ****************************************************
66 00008C                 *      switch analog output A off                  *
67 00008C                 ****************************************************
68 00008C 0000            AOFF   VO                          hyperprocessor off *
69 00008E 4E4F                  OFF                         to supervisor mode *
70 000090 13FC0007FFFF
        B800                    MOVE.B #7,PSGA              select register 7 *
71 000098 103FFFFFB800          MOVE.B PSGA,D0              get control register *
72 00009E 08C00000              BSET   #0,D0                switch A off      *
73 0000A2 13C0FFFFB802          MOVE.B D0,PSGD              save control register *
74 0000A8 027CDBFF              ANDI   #$DBFF,SR            allow interrupts  *
75 0000AC 4EEB0002              JMP    2(A0)                return to caller  *
76 0000B0                 ****************************************************
77 0000B0 R0000000              END    OF                   JOY               *
```

Die Assemblerroutinen sind erforderlich, um den Soundchip des Atari ST im Supervisor-Modus ansprechen zu können.

Workspace zurück. Auf dieselbe Weise kann mit VOLA die Lautstärke eingestellt werden. Da nur die letzten 4 Bit die Lautstärke bestimmen, wird von dem übergebenen Parameter nur die untere Hälfte in den Soundchip geschrieben.

Wenn sichergestellt wäre, daß nur eine Task die beiden Prozeduren anspricht, hätte eine Workspace-Definition gereicht. Durch die doppelte Workspace-Definition sind die Prozeduren aber vollkommen unabhängig und 'reentrant'; es könnten also beliebig viele Tasks am Sound 'mitmischen'.

Etwas anders sehen die Prozeduren AON und AOFF zum An- und Abschalten des Analogausgangs A aus, da sie keinen Workspace brauchen. Deshalb können sie gleich mit dem Ausschalten des Hyperprozessors beginnen. A0 zeigt beim Aufruf auf den Anfang der Parameterliste auf Aufruferseite. Da bei leerer Parameterliste dort nur ein EPAR steht und auch kein Workspace zurückzugeben ist, kann der Rücksprung in Zeile 64 beziehungsweise 75 auf die übernächste Adresse erfolgen.

I/O-Fehler? Kein Problem!

Das zugehörige Demonstrationsprogramm in PEARL stellt eine Einbau-Funktion vor, mit deren Hilfe der Programmierer den Status der Datenstationen abfragen und auf Besonderhei-

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /*   M U S I C   B Y   P E A R L                                 */
 4 /*   last update:  23.10.86   16h25                              */
 5 /*                                                               */
 6 /*****************************************************************/
 7 S=24;                          /* to spare memory */
 8 MODULE SOUND;
 9
10 SYSTEM;
11    A1;
12    DATA: ED.DUMMYNAME(NE);
13
14 PROBLEM;
15    SPC A1     DATION INOUT ALPHIC CONTROL(ALL);
16    SPC DATA   DATION IN    ALPHIC CONTROL(ALL);
17    SPC FREQA  ENTRY GLOBAL;
18    SPC VOLA   ENTRY GLOBAL;
19    SPC AON    ENTRY GLOBAL;
20    SPC AOFF   ENTRY GLOBAL;
21    DCL MSEC INV DUR INIT(0.001 SEC);
22
23 /*** MUSIK ***************************************************/
24 /*                                                               */
25 /*   Wir machen Musik, da geht dir der Hut hoch ...              */
26 /*                                                               */
27 /*****************************************************************/
28 MUSIK:TASK;
29    DCL NAME   CHAR(8);
30    DCL FREQ   FIXED;          /* frequency */
31    DCL VOLUME FIXED;          /* volume */
32    DCL TIME   FIXED;          /* duration of tone */
33
34    PUT 'Music by RTOS'  TO A1 BY SKIP,A;
35    PUT '-------------'  TO A1 BY SKIP,A;
36    PUT 'Melody: '  TO A1 BY (2)SKIP,A;          /* get data file name */
37    GET NAME FROM A1 BY SKIP,A;
38
39    OPEN DATA BY IDF(NAME);                          /* open data file */
40    CALL REWIND(DATA);
41    GET FREQ,VOLUME,TIME FROM DATA BY SKIP,(3)F(8);
42
43    CALL AON;                                /* switch analog output on */
44    WHILE ST(DATA) EQ 0 REPEAT          /* repeat until end of data file */
45       CALL FREQA(FREQ);                                /* set frequency */
46       CALL VOLA(VOLUME);                                   /* set volume */
47       AFTER TIME * MSEC RESUME;                                  /* wait */
48       GET FREQ,VOLUME,TIME FROM DATA BY SKIP,(3)F(8);
49    END;
50    PUT 'sorry, it''s done ...' TO A1 BY (2)SKIP,A,SKIP;
51    CALL AOFF;                                /* switch analog output off */
52 END;  /* of task MUSIK */
53 MODEND;

MODULE-SUMMARY:

TASKS:
(INT)MUSIK
PROCEDURES:
(ESR)AOFF        (ESR)AON        (ESR)VOLA        (ESR)FREQA
```

Das Demoprogramm spielt eine kleine Melodie ab.

200	13	150
200	11	10
178	13	150
178	11	10
159	13	150
159	11	10
150	13	150
150	11	10
133	13	310
133	11	10
133	13	310
133	11	10
119	13	150
119	11	10
119	13	150
119	11	10
119	13	150
119	11	10
119	13	150
119	11	10
133	13	630
133	11	10
119	13	150
119	11	10
119	13	150
119	11	10
119	13	150
119	11	10
119	13	150
119	11	10
133	13	630
133	11	10
150	13	150
150	11	10
150	13	150
150	11	10
150	13	150
150	11	10
150	13	150
150	11	10
159	13	310
159	11	10
159	13	310
159	11	10
133	13	150
133	11	10
133	13	150
133	11	10
133	13	150
133	11	10
133	13	150
133	11	10
200	13	630
200	12	30
200	11	20
200	10	10
200	9	10
200	7	10
200	4	10

Statt Noten: Datenliste einer bekannten Melodie. In den drei Spalten stehen Tonhöhe, Lautstärke und Tonlänge.

ten reagieren kann. 'ST (dationname)' heißt diese Funktion, die einen FIXED(15)-Wert liefert. Ist dieser Wert 0, so ist alles in Ordnung. Wird aber zum Beispiel eine 1 zurückgegeben, so ist ein File zu Ende, und es konnten keine Daten mehr gelesen werden. Die ausführliche Auflistung der Rückgabewerte findet sich in dem RTOS-Handbuch im Abschnitt C-III. In meinem Beispielprogramm benutze ich die Funktion, um das Ende der Eingabedatei zu erkennen. MUSIK erwartet ein ED-File, in dem FIXED-Werte stehen. Jeweils drei werden dann als Frequenz, Lautstärke und Dauer eines Tones interpretiert. Dank der Endeerkennung durch die ST-Funktion unterliegt die Eingabedatei keiner Längenbeschränkung (außer

durch die Größe des verfügbaren Speichers).

Trotz Statusabfrage werden aber bei I/O-Fehlern zusätzlich Fehlermeldungen ausgegeben. Dies kann man verhindern, indem man für die Datenstation im SYSTEM-Teil 'NE' (no error message) definiert (Zeile 12).

Ein kleiner Trick vereinfacht die Angabe der Zeitdauer: Ich definiere in Zeile 21 eine Zeitdauerkonstante von einer Millisekunde, die dann mit FIXED-Werten multipliziert werden kann (Zeile 47). Dadurch erscheinen die FIXED-Werte wie Zeitdauerangaben in Millisekunden.

Der Tonhöhenwert bezieht sich, wie schon erwähnt, nicht auf die Frequenz, sondern auf die Periodendauer des Tonsignals, also auf den Kehrwert der Frequenz. Aus dem Physikunterricht ist Ihnen gewiß noch die Regel geläufig, wonach die Tonhöhe sich um einen Halbton nach oben ändert, wenn der Wert der Frequenz mit der 12ten Wurzel aus 2 multipliziert wird. In unserem Fall ist der Kehrwert dieses Faktors zu verwenden. Durch wiederholtes Multiplizieren lassen sich somit alle Intervalle berechnen. Zwei Töne gleicher Tonhöhe kann man gegeneinander abgrenzen, indem man den ersten Ton teilt und den zweiten, kürzeren Teil etwas leiser erklingen läßt.

Übrigens: natürlich läuft die Musik weiter, auch wenn man im Editor arbeitet oder den Compiler startet. Dazu muß nur die Task MUSIK mit hinreichend hoher Priorität aufgerufen werden, etwa mit

MUSIK PRIO 1

Dann stört auch die Ausgabe eines Listings nicht. Nur Systemausgaben mit noch höherer Priorität durch Kommandos wie S oder L würden die Musik auf einem Ton stehenbleiben lassen, bis der Bildschirm wieder zur Ruhe gekommen ist. Aber wen interessiert schon der Systemstatus, wenn dadurch so wunderschöne Musik unterbrochen würde . . .

Nachtrag zu Teil 6

Obwohl es auf den ersten Blick so aussehen mag, ist die Assembler-Prozedur ALLOC nicht wiedereintrittsfähig. In der Modulvariablen MEMBLK kann nämlich immer nur die Adresse eines Speicherblockes stehen. Außerdem könnte ein erster Speicherblock nicht mehr angesprochen werden, wenn ein zweiter angefordert wird und damit die Feldköpfe von DATEI und ORDER Verweise auf den zweiten Speicherblock enthalten. In dem abgedruckten Programm ist dieses Verhalten unkritisch, da in dem Hauptprogramm ALLOC nicht zweimal aufgerufen wird, ohne daß zwischendurch mit FREE der Speicher wieder freigegeben würde.

Benchmarks unter RTOS

Beste Voraussetzungen zur Dokumentation der eigenen Leistungsfähigkeit bringt RTOS durch seine Echtzeitfähigkeit mit sich. Mittels der Systemuhr und der Datentypen Uhrzeit und Zeitdauer ist es sehr einfach möglich, die Rechenzeit beliebiger Operationen zu ermitteln. Soll die Rechenzeit für Programmteile ermittelt werden, läßt sich dies mittels des abgedruckten Programmrumpfes erreichen, indem der Programmteil innerhalb der REPEAT-Schleifen eingesetzt wird.

Natürlich müssen nicht 1000 × 1000 Schleifen durchlaufen werden. Auf jeden Fall ist aber die Zeit für die Schleifen 'leer' nachzumessen, um diesen Wert abziehen zu können. In der abgedruckten Form läuft das Programm etwa 7,5 Sekunden, woraus sich 7,5 Mikrosekunden für jeden Schleifendurchlauf ergeben. Bedenkt man, daß währenddessen etwa 1900mal ein Timer zum Weiterstellen der Systemuhr aufgefordert hat, so zeugt dies von schnellem Code.

Will man die Rechenzeit einer Task bestimmen, braucht man nur eine zweite Task mit niedrigerer Priorität zu schreiben, die die Uhrzeit abliest, die erste Task aktiviert und ein zweites Mal die Uhrzeit abliest. Dies funktioniert natürlich nicht mehr, wenn die erste Task irgendwelche Wartezu-

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /*  B E N C H M A R K   T E S T                                  */
 4 /*  last update: 27.07.86    15h30                               */
 5 /*                                                               */
 6 /*****************************************************************/
 7 $=200;
 8 MODULE BENCH;
 9 SYSTEM;
10    UHR: CL;
11    A1;
12
13 PROBLEM;
14    SPC UHR DATION IN    CLOCK;
15    SPC A1  DATION INOUT ALPHIC CONTROL(ALL);
16
17 TEST:TASK;
18    DCL (T1,T2) CLOCK;
19    DCL DAUER   DUR;
20
21    PUT 'ACHTUNG: Benchmark Test '' TO A1 BY SKIP,A,SKIP;
22    READ T1 FROM UHR;
23    TO 1000 REPEAT
24       TO 1000 REPEAT;
25       /* anything to measure */;
26       END;
27    END;
28    READ T2 FROM UHR;
29    DAUER = T2 - T1;
30    PUT T1,T2,DAUER TO A1
31     BY (2)(SKIP,T(12,3),(2)X),SKIP,D(23,3),SKIP;
32 END;
33 MODEND;

MODULE-SUMMARY:

TASKS:
(INT)TEST
     0 ERRORS. SIZE(16):0001BE
```

stände hat und die zweite wieder zum Laufen kommt, bevor die erste fertig ist. Aber in diesem Fall macht eine Zeitmessung ohnehin keine Aussage über die Laufzeit einer Task.

Programme in Eproms

In der c't-Ausgabe 2/86 wurde mit dem EPAC-68008 ein weiterer Einplatinen-Computer vorgestellt, der sich für Steuerungs- und Regelungsaufgaben gut eignen dürfte. Dies gilt vor allem deshalb, weil dem Winzling ein komplettes RTOS-Betriebssystem zur Verfügung steht, so daß er in PEARL programmiert werden kann. Als komfortable Entwicklungssysteme bieten sich dabei natürlich 'ausgewachsene' RTOS/PEARL-Rechner wie der c't 68000 oder der Atari ST an.

Das Werkzeug, das der Programmierer benötigt, um PEARL-Programme für den EPAC-68008 oder zur Erweiterung des eigenen Systems in EPROMs unterzubringen, steht deshalb im Mittelpunkt dieses Kapitels. Es handelt sich um den Befehl 'PROM', der im Handbuch beschrieben wird, bisher als Shell-Erweiterung verfügbar war und künftig zum Standard-Lieferumfang gehört. RTOS stellt mit diesem Befehl ein Mittel zur Verfügung, PEARL-Programme für das Einbrennen in EPROMs vorzubereiten. Außerdem werden die Kern-Routinen zur Steuerung des PROMMER 520 unter RTOS vorgestellt.

Um zu erklären, wie der PROM-Befehl arbeitet, muß ich etwas weiter ausholen: Im Abschnitt E-VII des Handbuchs, aber auch sonst ab und zu, taucht in Verbindung mit RTOS der Begriff 'Scheibe' oder 'Scheibenstruktur' auf. Dies deutet bereits darauf hin, daß RTOS nicht 'aus einem Guß' ist, sondern aus vielen Einzelteilen besteht. Diese Einzelteile liegen bei ausgeschaltetem Rechner ziemlich beziehungslos nebeneinander im Speicher. Erst mit dem Einschalten und dem Aufruf des sogenannten 'Nukleus' (Kern) bekommen sie Verbindung. Dazu muß der Nukleus natürlich die einzelnen Teile finden können. Als Kennung für den Nukleus hat deshalb jede Scheibe eine Signalmarke, die mit $AEB1, $BF95 beginnt. An diesen vier Bytes erkennt der Nukleus, daß dort ein Betriebssystemteil liegt.

Die nächsten beiden Bytes geben dann Auskunft darüber, welcher Art die gefundene Scheibe ist. Diese beiden Bytes berechnen sich nach der Formel $(x * 2 + 1) * 37$, wobei x die Nummer der Scheibe ist. $x = 1$ etwa kennzeichnet Scheiben, die Systemtasks enthalten, $x = 9$ steht für Scheiben, die Datenstationen definieren, und so weiter. Der Nukleus merkt sich in Tabellen, wo er die einzelnen Scheiben gefunden hat; über die

Tabellen sind die einzelnen Scheiben nun verbunden. Durch diesen Aufbau ist es relativ einfach, zum Beispiel neue Datenstationen oder neue Shell-Kommandos, wie etwa den PROM-Befehl, anzufügen.

Hinter der Nummer 13 verbirgt sich nun eine Scheibe, die Blöcke für Modulvariable definiert. Dies sind Speicherbereiche, die bei der Initialisierung des Systems an einer festen Stelle im Speicher eingerichtet werden, um gemeinsame Variable von ROM- oder EPROM-residenten Tasks aufzunehmen.

Allerdings müssen diese Speicherbereiche nicht zwingend Variable enthalten. In meinem Beispiel enthalten sie ein Modul und eine Task.

Dieses Modul mit seiner Task wird bei der Initialisierung mit dem beschriebenen Mechanismus im Speicher eingerichtet. Die Task läuft gleich nach der Initialisierung an, das heißt, natürlich erst nach den anderen Autostart-Tasks im System, die eine höhere Priorität haben. Zur Autostart-Task wurde die Task durch den Befehl AUTO-START, der sich mit dem PROM-Befehl gemeinsam in der Shell-Erweiterung befindet. Um das Modul mit der Task 'brennbar' zu machen, ist es ganz normal in den Speicher zu laden. Dabei sollte man allerdings schon darauf achten, wo es zu liegen kommt, denn an genau dieselbe Stelle wird es bei der Initialisierung auch wieder hingelegt. Für dieses Beispiel ist die Adreßlage allerdings unwichtig, da gleich als zweiter Befehl an den Kommandoprozessor das Entladen des eigenen Moduls angeordnet wird. Deshalb ist auch von Modul und Task nichts mehr zu spüren, wenn das System hochgefahren ist.

Danach gibt man den Befehl PROM und eine Liste von Tasks und/oder Modulen an, die in '13er Scheiben' umgewandelt werden sollen. Da der PROM-Befehl in die Standardausgabe des Bedien-Interface schreibt, ist diese mit dem O-Befehl in ein File oder direkt zu dem EPROM-Programmiergerät umzuleiten. Wird hinter einem Modulnamen ein '*' eingegeben, so werden alle Tasks, die direkt im Anschluß an dieses Modul stehen, auch in '13er Scheiben' umgewandelt. Wie die Scheiben später im EPROM aussehen, ist aus dem abgedruckten Speicherauszug zu ersehen.

Für Nachahmer noch ein paar Hinweise: der PROM-Befehl schickt vor der ersten Zeile des S-Records noch einige Nullen, um EPROMMER, die kein Handshake haben, nicht zu überrennen. Sollte ein Gerät diese Nullen nicht mögen, lassen sie sich leicht mit dem Editor entfernen (vorausgesetzt, man hat die Ausgabe in ein ED-File umgelenkt).

Die abgedruckte Autostart-Task dient lediglich zur Demonstration. Besitzer der neuesten EPROM-Version für den Atari ST (Version 1.1) finden bereits eine gleichartige Autostart-Möglichkeit vor. Bei der Boot-Diskette (Version C) ist diese allerdings nicht implementiert. Sie wäre wegen der unterschiedlichen Diskettenformate unter TOS und RTOS nicht sinnvoll zu nutzen.

Durch die Task AUTO wird nur ein einziger Befehl (außer dem zum Entladen des eigenen Moduls) an den Kommandoprozessor abgesetzt, nämlich der, den Inhalt des Files INIT im Laufwerk F0. in den Kommandokanal zu kopieren. Damit ist man nun sehr flexibel, da das File INIT beliebige Befehle enthalten kann.

EPROMs brennen unter RTOS

Als Beispiel für eine Interruptroutine unter RTOS beschreibe

Das kleine Beispielprogramm etabliert eine Autostart-Möglichkeit.

```
 1 /!!!
```

ich nun eine Prozedur zum Brennen von EPROMs mit dem PROMMER 520 aus c't 7/86. Das Besondere an dieser Prozedur ist, daß sie die verbleibende Rechenleistung des ATARI voll dem Nutzer zur Verfügung stellt.

Das Beispiel bezieht sich auf die Programmierung des EPROM-Typs 27256. Dieser Typ wird für die ST-EPROM-Bank und den ST-Userport (c't 3/86) verwendet. Wegen der unterschiedlichen Pinbelegung muß deshalb die Software angepaßt werden, wenn ein anderer EPROM-Typ verwendet wird. Dem Programmieralgorithmus liegt das Datenblatt des 27256 von Intel zugrunde. Dieser unterscheidet sich aber nur in der Berechnung der Nachbrenndauer und der Detektierung eines 'device fail' geringfügig von denen anderer Hersteller und wurde willkürlich gewählt.

Da der PROMMER 520 keine eigene Intelligenz (sprich: Mikroprozessor) besitzt, muß die gesamte Steuerung über den Hauptprozessor laufen. Dies wird erschwert durch den Umstand, daß nur jeweils ein 16-Bit-Eingabe- und 16-Bit-Ausgabeport vorhanden sind. So müssen das zu programmierende Datum, die Adresse und die Steuersignale nacheinander in Latches geschrieben werden. Der PROMMER besitzt ein Steuerregister, das durch das Latch 74LS273 realisiert ist. In dieses Register wird geschrieben, indem das gewünschte Byte auf die untere Hälfte des Ausgabeports gegeben wird. Mit einem Taktimpuls, wie er für serielle Erweiterungen des Userports vorgesehen ist, wird dieser Wert in das Steuerregister übernommen. Dabei ist es egal, ob ein '0'-Impuls ($FAFFFE) oder ein '1'-Impuls ($FAFFFD) verwendet wird. Der Inhalt des Steuerregisters kann nicht gelesen werden. Nur die Bits 5, 6 und 7 können über das Statusregister des Userports ($FAFFFF) abgefragt werden. Deshalb sollte man sich irgendwo eine Kopie des Registerinhalts ablegen. Die Bedeutung der einzelnen Bits ist im Kasten näher erläutert.

Bit	Bedeutung
0	Enable für Latch-Ausgänge
1	Vpp an Pin 1
2	Vpp an Pin 22
3	Vcc (5 V) an Pin 28
4	$\overline{OE}$ (Pin 22)
5	Vpp 12,5 V/21 V
6	Vcc (5 V) an Pin 26
7	Vcc (6 V) an Pin 28

Bit 0 ist direkt mit dem Enable-Eingang des Latches 74LS374 mit Tri-State-Ausgängen verbunden. Da dieser Eingang invertiert, bewirkt eine '1' in diesem Bit, daß die Ausgänge des Latches hochohmig sind. Dies ist erforderlich, wenn man Daten aus einem EPROM im Programmiersockel lesen will.

Eine '1' in Bit 1 schaltet die gewählte Programmierspannung auf Pin 1 des Programmiersockels. Dort wird sie bei 28poligen EPROMs benötigt. Mit Bit 2 kann die Programmierspannung an Pin 22 gelegt werden, wie es für 24polige EPROMs notwendig ist. Es darf aus verständlichen Gründen immer nur eines dieser beiden Bits '1' sein. Die Auswahl der Programmierspannung geschieht mit Bit 5. Dabei schaltet eine '0' auf 21V und eine '1' auf 12,5V.

Bit 3 und Bit 7 bilden ebenfalls ein sich ausschließendes Paar. Mit Bit 3 wird die Versorgungsspannung an Pin 28 für 28polige EPROMs auf 5V gesetzt, mit Bit 7 auf 6V. Für 24polige EPROMs wird mit Bit 6 eine Versorgungsspannung von 5V auf Pin 26 geschaltet.

Bit 4 schließlich stellt den $\overline{OE}$-Eingang (Output Enable) für 28polige EPROMs dar. Hierbei ist wieder zu beachten, daß dieser Eingang invertiert.

In das schon erwähnte Daten-Latch wird genau wie in das Steuerregister geschrieben – mit dem Unterschied, daß hier das LATCHIN-Signal ($FAFFF6) zur Übernahme verwendet wird.

So sieht die Autostart-Scheibe im EPROM aus.

```
ADR FB8080 120
FB8080: 0000 0000 0000 0000 0010 4120 2020 2020  ..........A.....
FB8090: AEB1 BF95 03E7 0007 0000 0007 0022 4141  .............."AA
FB80A0: 4141 4141 0007 000B 0000 000D 0010 4155  AAAA..........AU
FB80B0: 544F 2020 0007 0014 4081 0900 0000 00B0  TO......@.......
FB80C0: 0000 5843 FEFE 0000 0000 FFFF FFFF FFFF  ..XC............
FB80D0: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ................
FB80E0: 0000 0000 0000 0000 0010 4120 2020 2020  ..........A.....
FB80F0: AEB1 BF95 03E7 0007 0022 0007 00D8 4141  .........."....AA
FB8100: 4141 4141 0007 002A 0000 0007 0041 4155  AAAA...*.....AAU
FB8110: 544F 2020 0030 0000 00FE 0007 0040 0000  TO...0......@..
FB8120: 0003 0030 0007 00AC 0007 00AC 0000 0036  ...0...l...l...6
FB8130: 4FEC 00B0 42AC 00B0 42AC 00A2 42AC 009E  O...B...B...B...
FB8140: 42AD FFFC 2C78 0BCA 4E4E 1B28 0000 FFBA  B....x..NN.(....
FB8150: 000C 7120 0010 002B 6F80 000B 2DB0 0030  ..q....(o...-..0
FB8160: 2B00 4E4E 1B28 0000 FF70 000C 7120 000C  +.NN.(...p..q...
FB8170: 0020 6F80 000B 2DB0 0016 2B00 4E41 434F  ..o...-...+.NACO
FB8180: 5059 2046 302E 494E 4954 3E58 432E FFFF  PY.F0.INIT>XC...
FB8190: 554E 4C4F 4144 2041 5554 4F2A 0000 0000  UNLOAD.AUTO*....
FB81A0: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ................
```

Für die Adreßleitungen am EPROM gibt es kein Latch, daher müssen die Adressen zuletzt angelegt werden. Auch dies geschieht durch Schreiben in den Ausgabeport. Im Gegensatz zu den beiden Latches werden hier aber alle 16 Bit benutzt, während vorher die oberen 8 Bit ohne Bedeutung waren. Bit 15 der Adresse verdient dabei besondere Beachtung. Es ist auf Pin 1 des Programmiersockels geführt, an dem bei 28poligen EPROMs auch die Programmierspannung anliegt. Soll aber nicht programmiert, sondern nur aus dem EPROM gelesen werden, so muß hier eine '1' stehen. Beim Lesen verlangen die EPROMs am Vpp-Pin die Versorgungsspannung. Durch einen Pull-Down-Widerstand geht Pin 1 bei anliegender '0' auf 0V.

Bleibt noch der $\overline{CS}$-Eingang (Chip Select) an Pin 20. Dieser wird über die Handshake-Leitungen RBUSY und RSTROBE des Userports bedient. Da die RSTROBE-Leitung als Eingang nicht direkt zugänglich ist, ist der LATCHOUT-Ausgang (SFAFFF7) mit ihr verbunden. Nach einen LATCHOUT-Puls geht somit die $\overline{CS}$-Leitung auf '0'. Durch Lesen des Eingangsport-Latches kann das Select-Signal zurückgesetzt werden.

Userport unter PEARL

Den Userport und somit auch den PROMMER kann der Programmierer auch von PEARL aus leicht ansprechen. Als Datenstationstyp 'BASIC' spezifiziert (nicht zu verwechseln mit einer verbreiteten Programmiersprache), lassen sich alle Register des Userports lesen. Für die Ausgabe wird ein spe-

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /*  Eproms lesen in PEARL                                        */
 4 /*                                                               */
 5 /*  last update:  01.12.86   15:00                               */
 6 /*                                                               */
 7 /*****************************************************************/cam*/
 8 MODULE M;
 9 SYSTEM;
10    A1;
11    POUT:  BU(0203)-(-);          /* port output */
12    RESET: BU(01FAFFF5)(-);       /* software reset */
13    LIN:   BU(01FAFFF6)(-);       /* latch in */
14    LOUT:  BU(01FAFFF7)(-);       /* latch out */
15    PIN:   BU(02FAFFFB)(-);       /* read input port */
16    PINL:  BU(02FAFFFA)(-);       /* read input latch */
17    IOCLK: BU(01FAFFFD)(-);       /* clock for serial I/O */
18    STAT:  BU(01FAFFFF)(-);       /* port status */
19
20 PROBLEM;
21    SPC A1    DATION INOUT ALPHIC CONTROL(ALL);
22    SPC POUT  DATION   OUT BASIC;
23    SPC RESET DATION IN    BASIC;
24    SPC LIN   DATION IN    BASIC;
25    SPC LOUT  DATION IN    BASIC;
26    SPC PIN   DATION IN    BASIC;
27    SPC PINL  DATION IN    BASIC;
28    SPC IOCLK DATION IN    BASIC;
29    SPC STAT  DATION IN    BASIC;
30
31    DCL ECTRL BIT(16);                  /* Epromer control register */
32
33
34 /*****************************************************************/
35 /*  procedure to read a single byte from an eprom                */
36 /*****************************************************************/
37 READBYTE: PROC (ADR BIT(16)) RETURNS (BIT(8));
38    DCL DUMMY BIT(8);
39    DCL BYTE BIT(16);
40
41    ADR = ADR OR '8000'B4;              /* pin 1 to 5V */
42    ECTRL = ECTRL AND 'FFEF'B4;         /* /OE low */
43    SEND ECTRL TO POUT;
44    TAKE DUMMY FROM IOCLK;
45    SEND ADR TO POUT;                   /* set address */
46    TAKE DUMMY FROM LOUT;               /* /CE low */
47    TAKE BYTE  FROM PINL;               /* read byte, /CE high */
48    ECTRL = ECTRL OR '0010'B4;          /* /OE high */
49    SEND ECTRL TO POUT;
50    TAKE DUMMY FROM IOCLK;
51    RETURN(BYTE SHIFT 8);
52 END;   /* READBYTE:PROC */
53
54 /*****************************************************************/
55 /*  task to call the READBYTE procedure for 16 subsequent bytes  */
56 /*****************************************************************/
57 TEST:TASK;
58    DCL DUMMY BIT(8);
59    DCL BYTE BIT(8);
60    DCL ADR BIT(16);
61
62    ECTRL = '0001'B4;                   /* software reset */
63    SEND ECTRL TO POUT;
64    TAKE DUMMY FROM IOCLK;
65
66    ECTRL = '8009'B4;                   /* pin 1,28 to 5V */
67                                        /* data latch disabled */
68    PUT 'Addr: ' TO A1 BY SKIP,A;
69    GET ADR FROM A1 BY SKIP,B4(4);
70    PUT ADR,': ' TO A1 BY SKIP,B4(4),A;
71    TO 16 REPEAT
72       BYTE = READBYTE(ADR);
73       PUT BYTE,' ' TO A1 BY B4(2),A;
74       ADR = TOBIT(TOFIXED(ADR) + 1);
75    END;
76    ECTRL = '0001'B4;                   /* pin 1,28 to 0V */
77    SEND ECTRL TO POUT;
78    TAKE DUMMY FROM IOCLK;
79 END;   /* TEST: TASK; */
80
81 /*****************************************************************/
82 /*  task to reset the userport                                   */
83 /*****************************************************************/
84 RESET: TASK;
85    DCL DUMMY BIT(8);
```

```
86
87     ECTRL = '0001'B4;                    /* pin 1,28 to 0V */
88     SEND ECTRL TO POUT;
89     TAKE DUMMY FROM IOCLK;
90 END;    /* RESET: TASK; */
91
92
93 MODEND;
```

Das Auslesen eines EPROMs mit dem PROMMER 520 läuft vollständig unter PEARL ab.

zielles Konstrukt benötigt, da diese ja über einen Lesebefehl geschehen muß. Darum muß sich aber der Programmierer nicht weiter kümmern, denn dieses Konstrukt wird selbständig vom Compiler erstellt. Der Ausgabeport, wie er mit Zeile 11 eingebunden ist, verhält sich also ganz normal.

Bei den Registern des Userports muß zwischen Byte- und Wortzugriffen unterschieden werden. Dazu dienen die ersten beiden Ziffern in der Klammer (Zeilen 12 bis 18). Steht dort '01', so wird byteweise, bei '02' wortweise gelesen.

Als Beispiel soll eine Prozedur dienen, mit der der Inhalt eines EPROMs im Programmiersockel ausgelesen werden kann. Für viele etwas gewöhnungsbedürftig dürfte der Umgang mit Bitfeldern in PEARL sein. In PEARL wird links mit 1 beginnend gezählt. Außerdem werden Bitfelder linksbündig abgelegt. Ein Bitfeld BIT(7) mit dem Inhalt '1001011'B liegt im Speicherwort als '9600'B4. Mit TO-FIXED wird dann daraus der FIXED(15)-Wert –27136.

Doch nun zu dem Beispielprogramm. Die Task TEST initialisiert als erstes den PROMMER. Der Reset-Befehl für den Userport ($FAFFF5) ist hierfür nicht so gut geeignet, da er zwar auch alle Spannungen zurücksetzt, die Ausgänge des Daten-Latches aber aktiviert. Hier vermißt man einen Inverter in der Enable-Leitung.

ECTRL dient in dem Programm als Kopie des Steuerregisters. Mit Zeile 63 schreibe ich diesen Wert in den Ausgabeport. Durch den Lesezugriff in Zeile 64 wird dieser Wert dann in das Steuerregister übernommen. Da der TAKE-Befehl nicht ohne Parameter auskommt, habe ich eine DUMMY-Variable eingerichtet. Diese Variable dient nur als Ziel für Lesezugriffe, ihr Inhalt besitzt keinen Aussagewert.

Mit den Zeilen 68 und 69 wird eine Adresse eingegeben, die als Eingangsparameter für die Funktion READBYTE dient. Diese gibt jeweils das gelesene Byte zurück, das hinter der Adresse auf den Bildschirm ausgegeben wird, bis 16 Bytes gelesen wurden. Die etwas merkwürdig anmutende Konstruktion in Zeile 74 ist notwendig, da für Bitfelder keine Addition definiert ist. Dabei funktioniert das Ganze nur deshalb, weil die Länge des Bitfeldes und die Länge des FIXED-Formates genau übereinstimmen.

In der Funktion READBYTE wird zunächst das Bit 15 der Adresse für den Vpp-Pin, wie schon oben beschrieben, auf '1' gesetzt. In Zeile 42 wird das $\overline{OE}$-Bit in ECTRL auf '0' gesetzt und anschließend ECTRL in das Steuerregister geschrieben. Mit Zeile 45 lege ich dann die Adresse an das EPROM und ziehe mit Zeile 46 den $\overline{CS}$-Eingang auf '0'. Dadurch ist das EPROM selektiert, und es legt den Inhalt der adressierten Speicherzelle an die Datenleitungen. Von dort lese ich den Inhalt mit Zeile 47 ein. Da ich aus dem Eingangs-Latch lese, wird gleichzeitig $\overline{CS}$ wieder '1' und das EPROM deselektiert. Schließlich wird mit den Zeilen 48 bis 50 noch $\overline{OE}$ wieder auf High gesetzt.

Um von den 16 Eingabeportbits die unteren acht als Funktionswert zurückzugeben, muß ich sie in die obere Worthälfte schieben. Die zugrundeliegende PEARL-Philosophie habe ich ja oben schon näher erläutert.

Programmieren im Hintergrund

Doch nun zum eigentlichen Kernstück. Ein Problem beim Brennen von EPROMs ist das programmgesteuerte Überwachen der Zeiten. Dafür habe ich den Timer C des Multi-Function-Peripheral-Bausteins (MFP) benutzt. Dieser Timer wird in der Prozedur so aufgesetzt, daß er nach etwa 960 Mikrosekunden einen Interrupt generiert (Zeilen 127 bis 141). Da sich die Peripheriebausteine im ST nur im Supervisor-Modus ansprechen lassen, schalte ich diesen in Zeile 126 ein. Ab Zeile 143 starte ich dann den ersten Programmierimpuls.

Jetzt müßte die Prozedur eigentlich auf das Verstreichen der ersten Millisekunde warten, bevor sie mit Verify und weiteren Programmierimpulsen die Programmierung fortsetzen könnte. Viele Programme drehen solange in einer Schleife ihre Runden. Um diese Rechenzeit anderweitig nutzen zu können, schicke ich die Prozedur in den Zeilen 155 bis 157 'schlafen'.

```
 1 000000                    ****************************************************
 2 000000                    *                                                  *
 3 000000                    *  PROCEDUR ZUR PROGRAMMIERUNG EINES BYTES         *
 4 000000                    *   EINES 27256 EPROMS                             *
 5 000000                    *                                                  *
 6 000000                    * last update:   01.12.86        15:00             *
 7 000000                    *                                                  *
 8 000000                    ****************************************************Ecm
 9 000000                    *        MODULE HEAD FOR RTOS-UH                   *
10 000000                    *                                                  *
11 000000 00000000                DC.L  0                  for loader            *
12 000004 00000000                DC.L  0                  for loader            *
13 000008 0010                    DC    $10                Type: MDLE            *
14 00000A 455058202020            DC.B  'EPX   '           module name           *
15 000010                    ****************************************************
16 000010                    *        SYSTEM TRAPS                              *
17 000010                    *                                                  *
18 000010                    GDPC  OPD   $4E43             call dispatcher       *
19 000010                    RETN  OPD   $4E4C             return from proc.     *
20 000010                    OFF   OPD   $46AF             interrupts off        *
21 000010                    VARW  OPD.V 10                FIXED(15) by ident    *
22 000010                    INVW  OPD.V 14                FIXED(15) by value    *
23 000010                    EPAR  OPD.V 19                end of para xfer      *
24 000010                    ENTR  OPD.V 29                procedure entry       *
25 000010                    ****************************************************
26 000010                    *        MEMORY SECTION DEFINITIONS                *
27 000010                    *                                                  *
28 000010 00000024           BLOCK EQU   $24               block-byte of a task  *
29 000010 00000004           BLKBSU EQU  4                 susp. bit no in BLOCK *
30 000010                    ****************************************************
31 000010                    *        ADDRESS DEFINITIONS                       *
32 000010                    *                                                  *
33 000010                    *        interrupt vector definitions              *
34 000010 00000114           TCVEC EQU   $114              *
35 000010                    *        userport definitions                      *
36 000010 00FAFFF5           UPCLR EQU   $FAFFF5           reset                 *
37 000010 00FAFFF6           UPLIN EQU   $FAFFF6           latch in puls         *
38 000010 00FAFFF7           UPLOUT EQU  $FAFFF7           latch out pulse       *
39 000010 00FAFFFB           UPPIN EQU   $FAFFFB           input port            *
40 000010 00FAFFFA           UPPINL EQU  $FAFFFA           input port latched    *
41 000010 00FAFFFD           UPIOC EQU   $FAFFFD           serial I/O clock      *
42 000010 00FAFFFA           UPSTAT EQU  $FAFFFA           userport status reg   *
43 000010 00F80000           UPOUT EQU   $F80000           output port base addr *
44 000010                    *        MFP 68901 definitions                     *
45 000010 00FFFA09           IERB  EQU   $FFFA09           MFP IR enable reg     *
46 000010 00FFFA15           IMRB  EQU   $FFFA15           MFP IR mask reg       *
47 000010 00FFFA1D           TCDCR EQU   $FFFA1D           MFP timer CD ctrl reg *
48 000010 00FFFA23           TCDR  EQU   $FFFA23           MFP timer C data reg  *
49 000010                    ****************************************************
50 000010                    *        SYSTEM VARIABLE DEFINITIONS               *
51 000010                    *                                                  *
52 000010 000007FE           IID   EQU   $7FE              ID act. running IR    *
53 000010 00000800           DPC   EQU   $800              dispatcher-call flag  *
54 000010 00000802           TID   EQU   $802              ID act. running task  *
55 000010 0000090A           DPCJ  EQU   $90A              jump to dispatcher    *
56 000010                    ****************************************************
57 000010                    *        PROCEDURE WORKSPACE DEFINITIONS           *
58 000010                    *                                                  *
59 000010 00000000           ADR   EQU   0                 addr where to program *
60 000010 00000002           BYTE  EQU   ADR+2             byte to program       *
61 000010 00000004           MODE  EQU   BYTE+2            programming mode      *
62 000010 0000000A           RTN   EQU   MODE+2            addr of return value  *
63 000010 0000000A           WSPSZ EQU   RTN+4             proc workspace size   *
64 000010                    ****************************************************
65 000010                    *        INTERRUPT BUFFER DEFINITIONS              *
66 000010                    *                                                  *
67 000010 00000000           OWNTID EQU  0                 own task ID           *
68 000010 00000004           PWSP  EQU   OWNTID+4          addr of proc workspace*
69 000010 00000008           CNT   EQU   PWSP+4            retry count           *
70 000010 0000000A           FLAGS EQU   CNT+2             flags                 *
71 000010 0000000C           ECTRH EQU   FLAGS+2           must be FF            *
72 000010 0000000D           ECTRL EQU   ECTRH+1           eprom ctrl reg        *
73 000010 0000000E           OLDVEC EQU  ECTRL+1           old IR vector         *
74 000010 00000012           IRBSZ EQU   OLDVEC+4          IR buffer size        *
75 000010                    IRBUF DS    IRBSZ             allocate memory       *
76 000022                    ****************************************************
77 000022                    *        MISCELLANEOUS DEFINITIONS                 *
78 000022                    *                                                  *
79 000022                    *        flag definitions                          *
80 000022 00000000           FINPLS EQU  0                 initial program puls  *
81 000022 00000001           FOVPRG EQU  1                 overprogram puls      *
82 000022                    *        mode definitions                          *
83 000022 00000000           MVPP  EQU   0                 0: 12,5V 1: 21V       *
84 000022 00000001           MVCC  EQU   1                 0: 5V,   1: 6V        *
85 000022 0000000C           TIME  EQU   12                timer data for 1 us   *
86 000022 00000019           MAXCNT EQU  25                max # of retries       *
87 000022                    ****************************************************
88 000022                    *        PROGRAM ONE BYTE OF AN EPROM              *
89 000022                    ****************************************************
90 000022                    *        REGISTER USAGE                            *
91 000022                    *        A2      address of interrupt buffer       *
92 000022                    *        A5      address of procedure workspace    *
93 000022                    *                                                  *
94 000022 1DC000000000A  \P256: ENTR  WSPSZ.L               fetch storage         *
95 000028 0E000000           INVW  ADR.X                   addr where to program *
```

Dazu setze ich im Statusbyte der aufrufenden Task das 'SUSPend'-Bit. Damit ist diese Task für den Dispatcher nicht mehr lauffähig.

Die Adresse des Statusbytes findet man über die Adresse des Task-Kopfes. Diese ist für jede laufende Task jeweils in der Speicherzelle TID ($802) abgelegt. Hierzu ist noch ein Offset von $24 zu addieren. Ist eine Millisekunde verstrichen, feuert Timer C einen Interrupt, und der Prozessor springt die Interruptroutine in Zeile 177 an. Dazu mußte in der Prozedur der zugehörige Interruptvektor 'verbogen' werden (Zeilen 131 bis 135), wobei das oberste Byte erhalten bleiben muß. Tritt nämlich in der Interruptroutine

ein Fehler auf, etwa ein Address Error, so kann das Betriebssystem anhand dieses Bytes herausfinden, über welchen Vektor die fehlerhafte Routine gefunden werden kann. Für diesen Fall muß der Programmierer unmittelbar vor der Interruptroutine den Offset zu einem 'Fehlerausgang' ablegen. Dieser Fehlerausgang muß dann dafür sorgen, daß gerettete Register geladen werden und der Stack aufgeräumt wird.

Ich setze außerdem den Timer zurück, so daß er keine weiteren Interrupts generieren kann. Dies gilt aber in dieser Form nur für Systeme mit dem 68000-Prozessor. RTOS läuft jedoch in der neuesten Version auch mit dem 68010 und 68020; deshalb habe ich auch die Fehlerbehandlung für diese angegeben. Da der 68020 alle 32 Bit als Adressen gebraucht (und nicht nur 24 wie der 68000), ist das obere Byte nicht verfügbar. Damit man trotzdem im Fehlerfalle herausfinden kann, welche Interruptroutine gerade lief, muß diese ihre Vektornummer in der Zelle IID ($7FE) eintragen, nachdem

```
96 00002C 0E000002          INVW   BYTE.X        byte to program      *
97 000030 0E000004          INVW   MODE.X        programming mode     *
98 000034 0A400006          VARW   RTN.Z         addr of return value *
99 000038 1300              EFAR                 end of para xfer     *
100 00003A 4A3900FAFFF5     TST.B  UPCLR         reset uservport      *
101 000040 45FAFFCE         LEA    IRBUF,A2      get addr of IR buffer *
102 000044 257B0802000      MOVE.L TID,DWNTID(A2) save own task ID     *
103 00004A 2540004          MOVE.L A5,PWSP(A2)    save addr of PWSP    *
104 00004E                  return, if BYTE = FF
105 00004E 0C2D00FF0003     CMPI.B =#FF,BYTE+1.X  check for equal      *
106 000054 660C             BNE.S  P1             bi is not equal to FF *
107 000056 323C000F         MOVE   =#$0FF,D1      code: no error       *
108 00005A 206D0006         MOVEA.L RTN.X,A0      xfer return value    *
109 00005E 3081             MOVE   D1,(A0)
110 000060 4E4C             RETN                  back to caller       *
111 000062                  prepare anything for programming
112 000062 7000             MOVEQ  =0,D0          clear D0             *
113 000064 25400008         MOVE   D0,CNT(A2)     initialize puls count *
114 000068 25400004         MOVE   D0,FLAGS(A2)   initialize flags     *
115 00006C 25400008         MOVE   D0,ECTRH(A2)   init eprommer ctrl reg*
116 000070 08EA0010000D     BSET   =1,ECTRL(A2)   Vpp at pin 1         *
117 000076 08EA0004000D     BSET   =4,ECTRL(A2)   /OE high             *
118 00007C 0B2D0000000D     BTST   =#VPP,MODE+1.X check program voltage *
119 000082 6606             BNE.S  P2             bi 21V selected      *
120 000084 08EA0005000D     BSET   =5,ECTRL(A2)   select Vpp = 12,5V   *
121 000088 0B2D0001000D     BTST   =#VCC,MODE+1.X check supply voltage *
122 00008C 6708             BEQ.S  P3             bi 5V selected       *
123 00008E 08EA0007000D     BSET   =7,ECTRL(A2)   select Vcc = 6V      *
124 000092 6006             BRA.S  P4
125 000094 08EA0003000D P3: BSET   =3,ECTRL(A2)   select Vcc = 5V      *
126 000040 4E4F             P4:    OFF            no irq allowed        *
127 000042 103900FFFA1D     MOVE.B TCDCR,D0       init timer C          *
128 000048 00000070         ORI.B  =#70,D0        /200 prescaler        *
129 00004C 13C000FFFA1D     MOVE.B D0,TCDCR
130 000052 13FC000C00FF
            FA23            MOVE.B =TIME,TCDR     set timer C data reg  *
131 00005A 45FB0114         LEA    TCVEC,A1       addr timer C vector   *
132 00005E 2551000C         MOVE.L (A1),OLDVEC(A2) save old IR vector   *
133 000062 47FA0082         LEA    EPIRQ,A3       get addr of IR routine*
134 000066 22B9             MOVE.L A3,(A1)        set new vector        *
135 000068 12AA000E         MOVE.B OLDVEC(A2),(A1) recover MSB of vector*
136 00006C 103900FFFA15     MOVE.B IMRB,D0        clear timer C IRQ mask*
137 000072 00000020         ORI.B  =#20,D0
138 00006A 13C000FFFA15     MOVE.B D0,IMRB
139 00006C 103900FFFA09     MOVE.B IERB,D0        enable timer C IRQ    *
140 000072 00000020         ORI.B  =#20,D0
141 00006A 13C000FFFA09     MOVE.B D0,IERB
142 000060                  start first program puls
143 00006C 08EA0000000A     BSET   =FINPLS,FLAGS(A2) first program puls *
144 000062 45FB00F80000     LEA    UPOUT,A1       base addr for output  *
145 00006B 7200             MOVEQ  =0,D1          clear D1              *
146 00006FA 12D0003         MOVE.B BYTE+1.X,D1    get data to program   *
147 00006FE 4A31B00         TST.B  0(A1,D1.L)     write to output port  *
148 000102 4A3900FAFFF6     TST.B  UPLIN          store at data latch   *
149 000108 12D0000C         MOVE   ECTRH(A2),D1   get eprom ctrl reg    *
150 00010C 4A31B00          TST.B  0(A1,D1.L)     write to output port  *
151 000110 4A3900FAFFFD     TST.B  UPIOC          store at ctrl register*
152 000116 12D0000         MOVE   ADR.X,D1        get program address   *
153 00011A 4A31B00          TST.B  0(A1,D1.L)     write to output port  *
154 00011E 4A3900FAFFF7     TST.B  UPLOUT         /CE low               *
155 000124 226A0000         MOVEA.L DWNTID(A2),A1 get own task ID       *
156 00012B 08E90004002A     BSET   =BLKBSU,BLOCK(A1) suspend itself     *
157 00012E 027C0BFF         ANDI   =#0BFF,SR      enable interrupts     *
158 000122 4E42             GOFC                  call dispatcher       *
159 000124                  wait for IR routine to wake this task
160 000124                  ----------------------------------------------
161 000124                  restore old IR vector
162 000124 4E4F             OFF                   no irq allowed        *
163 00013a 45FB0114         LEA    TCVEC,A1       addr timer C vector   *
164 00013A 22AA000E         MOVE.L OLDVEC(A2),(A1) restore old IR vector*
165 00013E 027C0BFF         ANDI   =#0BFF,SR      enable interrupts     *
166 000142 4E4C             RETN                  back to caller        *
167 000144
168 000144  ****************************************************************
169 000144                  INTERRUPT ROUTINE
170 000144  ****************************************************************
171 000144                  REGISTER USAGE
172 000144                  A1:    output port base address
173 000144                  A2:    address of interrupt buffer
174 000144                  A3:    address of procedure workspace
175 000144
176 000144 01C8             DC     IRMAL-EPIRQ    malfunction link
177 000146 3F3B07FE     EPIRQ MOVE  IID,-(A7)     save old IR ID
178 00014A 31FC0114407FE    MOVE   =TCVEC,IID     save own ID
179 000150 48E7E0E4         MOVEM.L D0-D2/A0-A2/A5,-(A7) save register
180 000154 45FAFEBA         LEA    IRBUF,A2       get addr of IR buffer
181 000158 2AAA0004         MOVEA.L PWSP(A2),A5   addr of proc workspace
182 00015C 47F900F80000     LEA    UPOUT,A1       base addr for output  *
183 000162 08ZA0000000A     BTST   =FINPLS,FLAGS(A2) first program puls *
184 00016B 6700008b         BEQ    EP2            bi not first pulses
185 00016C                  verify just now programmed byte
186 00016C 4A3900FAFFFA     TST.B  UPPIN          /CE high
187 000172 08EA0000000D     BSET   =0,ECTRL(A2)   disable data latch
188 00017B 08AA0004000D     BCLR   =4,ECTRL(A2)   /OE low
```

sie die vorher dort stehende Nummer auf den Stack gerettet hat. Somit sind die Zeilen 177, 178, 251 und 293 nur für Besitzer der Update-Version notwendig.

An dieser Stelle möchte ich noch einmal ganz ausdrücklich darauf hinweisen, daß RTOS in gewissen Grenzen Fehler in Interruptroutinen zuläßt. Es meldet sich also nicht gleich ab, sondern hat hierfür einen geordneten Rückfallmechanismus.

In der Interruptroutine hole ich mir die Adresse des Interruptpuffers, in dem ich vorher (in Zeile 103) auch die Adresse des Prozedur-Workspace abgelegt hatte. Dieser Interruptpuffer ist in den Zeilen 65 bis 75 definiert. Es werden dann solange weitere 1-ms-Programmierimpulse aufgesetzt, bis der geschriebene Wert auch wieder gelesen werden kann. Durch Multiplizieren der Anzahl der Programmierimpulse bis zum ersten geglückten Verify mit 3 Millisekunden bekommt man die Dauer des Impulses, mit dem nachgebrannt werden muß. Dabei muß der Timer eventuell mehrmals aufgesetzt werden, wenn die Nachbrenndauer groß ist.

Wenn das Byte fehlerfrei programmiert ist oder vorher ein Fehler auftrat, wird ab Zeile 277 der Timer zurückgesetzt. Ab Zeile 289 wird dann die schlafende Task wieder geweckt, indem das 'SUSPend'-Bit zurückgesetzt wird. Der abschließend aufgerufene Dispatcher teilt dann der Task wieder den Prozessor zu, wenn nicht noch Aktivitäten höherer Priorität im Gange sind.

Als Funktionswert wird entweder das programmierte Byte oder eine negative Fehlernummer zurückgegeben. Folgende

```
189 00017E 322A000C          MOVE   ECTRH(A2),D1    get eprom ctrl reg   !
190 000182 4A311800          TST.B  0(A1,D1.L)      write to output port  !
191 0001BA 4A3900FAFFFD      TST.B  UPIOC           store at ctrl register!
192 0001BC 322D0000          MOVE   ADR.X,D1        get program address   !
193 000190 4A311800          TST.B  0(A1,D1.L)      write to output port  !
194 000194 4A3900FAFFF7      TST.B  UPLOUT          /CE low               !
195 00019A                   do some stuff before reading the input port !
196 00019A 08BA000000D0      BCLR   #0,ECTRL(A2)    enable data latch     !
197 0001A0 08EA00040000      BSET   #4,ECTRL(A2)    /OE high              !
198 0001A6 303900FAFFF8      MOVE   UPPIN,D0        read input port       !
199 0001AC 4A3900FAFFFA      TST.B  UPPINL          /CE high              !
200 0001B2 B02D0003          CMP.B  BYTE+1.X,D0     test for equal        !
201 0001B6 6730              BEQ.S  EP1             bi verify ok          !
202 0001B8                   check for device fail                        !
203 0001B8 74FF              MOVEQ  #-1,D2          err code: device fail !
204 0001BA 52A6000B          ADDQ   #1,CNT(A2)      incr retry count      !
205 0001BE 0CA00190000B      CMPI   #MAXCNT,CNT(A2) to often?             !
206 0001C4 670000F2          BEQ    EX2             bi bad device, exit   !
207 0001C8                   start next program puls                      !
208 0001CB 322A000C          MOVE   ECTRH(A2),D1    get eprom ctrl reg    !
209 0001CC 4A311800          TST.B  0(A1,D1.L)      write to output port  !
210 0001D0 4A3900FAFFFD      TST.B  UPIOC           store at ctrl register!
211 0001D6 322D0000          MOVE   ADR.X,D1        get program address   !
212 0001DA 4A311800          TST.B  0(A1,D1.L)      write to output port  !
213 0001DE 4A3900FAFFF7      TST.B  UPLOUT          /CE low               !
214 0001E4 60000082          BRA    EP9             exit                  !
215 0001E8                   !-----------------------------------------!
216 0001E8                   !                                         !
217 0001E8                   !                                         !
218 0001EB 342A0008          EP1:   MOVE   CNT(A2),D2      get counter    !
219 0001EC C4FC000C          MULU   #TIME,D2        calc timer ticks      !
220 0001F0 C4FC0003          MULU   #3,D2            for overprogram puls !
221 0001F4 35420008          MOVE   D2,CNT(A2)      save the result       !
222 0001F8 08BA000000D0      BCLR   #FINPLS,FLAGS(A2) start of overprog   !
223 0001FE 08EA0001000A      BSET   #FOVPRG,FLAGS(A2)                     !
224 000204                   !  start overprogram puls                    !
225 000204 322A000C          MOVE   ECTRH(A2),D1    get eprom ctrl reg    !
226 000208 4A311800          TST.B  0(A1,D1.L)      write to output port  !
227 00020C 4A3900FAFFFD      TST.B  UPIOC           store at ctrl register!
228 000212 322D0000          MOVE   ADR.X,D1        get program address   !
229 00021A 4A311800          TST.B  0(A1,D1.L)      write to output port  !
230 00021A 4A3900FAFFF7      TST.B  UPLOUT          /CE low               !
231 000220                   !  manage overprogram puls                   !
232 000220                   !                                         !
233 000220 7400              EP2:   MOVEQ  #0,D2            code: no error !
234 000222 082A0001000A      BTST   #FOVPRG,FLAGS(A2) overprogram done?   !
235 000228 6756              BEQ.S  EX2             bi it's done, exit    !
236 00022A 103900FFFA1D      MOVE.B TCDCR,D0        stop timer C          !
237 000230 020000DF          ANDI.B #$DF,D0                               !
238 000234 13C000FFFA1D      MOVE.B D0,TCDCR                              !
239 00023A 302A0008          MOVE   CNT(A2),D0      get counter           !
240 00023E 04400100          SUBI   #$100,D0        more than 256 ticks?  !
241 000242 6A0A              BPL.S  EP3             bi yes, more          !
242 000244 08BA0001000A      BCLR   #FOVPRG,FLAGS(A2) end of overprogram  !
243 00024A 342A0008          MOVE   CNT(A2),D2      get counter           !
244 00024E 35420008          MOVE   D2,CNT(A2)      save the rest         !
245 000252 13C200FFFA23      MOVE.B D2,TCDR         set timer C data reg  !
246 000258 103900FFFA1D      MOVE.B TCDCR,D0        start timer C         !
247 00025E 00000070          ORI.B  #$70,D0         /200 prescaler        !
248 000262 13C000FFFA1D      MOVE.B D0,TCDCR                              !
249 000268                   !  restore registers, wait for next timer IR !
250 000268 4CDF2707          EP9:   MOVEM.L (A7)+,D0-D2/A0-A2/A5 recover regs !
251 00026C 31DF07FE          MOVE   (A7)+,IID       recover old IID       !
252 000270 007C0700          EX1:   ORI   #$0700,SR         all IR off    !
253 000274 4A7B0800          TST    DPC             dispatcher call?      !
254 000278 4B02              BMI.S  EX1             bi no proc. switching !
255 00027A 4E73              RTE                    normal exit           !
256 00027C 4EFB0906          EX1:   JMP     DPCJ            call dispatcher !
257 000280                   !-----------------------------------------!
258 000280                   !  verify programmed byte with Vcc = Vpp = 5V !
259 000290 7209              EX2:   MOVEQ  #$09,D1          disable data latch !
260 000282 4A311800          TST.B  0(A1,D1.L)      write to output port  !
261 000286 4A3900FAFFFD      TST.B  UPIOC           store at ctrl register!
262 00028C 322D0000          MOVE   ADR.X,D1        get program address   !
263 000290 00418000          ORI    #$8000,D1       set MSB               !
264 000294 4A311800          TST.B  0(A1,D1.L)      write to output port  !
265 000298 4A3900FAFFF7      TST.B  UPLOUT          /CE low               !
266 00029E 7400              MOVEQ  #0,D2           clear D2              !
267 0002A0 4E71              NOP                    wait                  !
268 0002A2 4E71              NOP                    !                     !
269 0002A4 343900FAFFF8      MOVE   UPPIN,D2        read input port       !
270 0002AA 4A3900FAFFFA      TST.B  UPPINL          /CE high              !
271 0002B0 B42D0003          CMP.B  BYTE+1.X,D2     test for equal        !
272 0002B4 6702              BEQ.S  EX3             bi device OK          !
273 0002B6 74FD              MOVEQ  #-3,D2          err code: device fail !
274 0002B8                   !-----------------------------------------!
275 0002B8                   !  disable timer C, wake sleeping task       !
276 0002B8                   !  D2:    return value                       !
277 0002B8 206D000A          EX3:   MOVEA.L RTW.X,A0        xfer return value !
278 0002BC 3082              MOVE   D2,(A0)                               !
279 0002BE 4A3900FAFFF5      TST.B  UPCLR           reset userport        !
280 0002C4 103900FFFA09      MOVE.B IERB,D0         disable timer C IRQ   !
281 0002CA 020000DF          ANDI.B #$DF,D0                               !
282 0002CE 13C000FFFA09      MOVE.B D0,IERB                               !
283 0002D4 103900FFFA15      MOVE.B IMRB,D0         mask timer C IRQ      !
```

Die Assemblerroutinen ermöglichen es, unter RTOS im Hintergrund EPROMs zu brennen.

Fehlernummern können auftreten:

–1: Byte läßt sich nach 25 Programmierpulsen noch nicht lesen

–2: Fehler innerhalb der Interruptroutine

–3: Fehler beim Verify bei abgeschalteter Programmierspannung

Es würde den Rahmen eines Artikels sprengen, ein komplettes EPROM-Programmierprogramm mit allem Komfort vorzustellen. Auch läßt sich sicher die Fehlerbehandlung stärker differenzieren. So wird zum Beispiel die vorgesehene Überstromerkennung bei falschem EPROM-Typ nicht berücksichtigt. Auch könnte man überprüfen, ob sich ein Byte überhaupt noch programmieren läßt, wenn vorher schon ein anderer Wert eingebrannt wurde. Das abgedruckte Programm sollte also lediglich als Grundlage und Anregung für Ihre eigene Programmentwicklung unter PEARL verstanden werden. ct

```
284 00020A 0200000DF          ANDI.B #$DF,D0                    *
285 00020E 13C000FFFA15       MOVE.B D0,IMRB                    *
286 000214 103900FFFA1D       MOVE.B TCDCR,D0        stop timer C *
287 00021A 0200000DF          ANDI.B #$DF,D0                    *
288 00021E 13C000FFFA1D       MOVE.B D0,TCDCR                   *
289 000224 20BA0000           MOVEA.L OWNTID(A2),A0 get own task ID *
290 00022B 08AB00040024       BCLR   #BLKBSU,BLOCK(A0) clear suspend bit *
291 000232 537B0800           SUBQ   #1,DPC          mark dispatcher call *
292 000302 4CDF2707           MOVEM.L (A7)+,D0-D2/A0-A2/A5 recover regs *
293 00030a 31DF07FE           MOVE   (A7)+,IID       recover old IID *
294 00030A 4EFB0906           JMP    DPCJ            call dispatcher *
295 00030E                  ------------------------------------------*
296 00030E                  *  provide malfunction exit               *
297 00030E 45FAFD00         IRMAL: LEA    IRBUF,A2    get addr of IR buffer *
298 000312 2BAA0004           MOVEA.L PWSP(A2),A5   addr of proc workspace*
299 00031a 74FE               MOVEQ   #-2,D2        fatal IR routine error*
300 000318 609E               BRA.S   EXT           disable IR, disp. call*
301 00031A                  ******************************************
302 00031A FFFFFFFF           END     OF            JOY               *
```

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /*  Einfaches Programm zum Aufruf der Eprom-Programmier-Prozedur  */
 4 /*                                                               */
 5 /*  last update:   1.12.86   15h00                               */
 6 /*                                                               */
 7 /*****************************************************************cmw*/
 8 MODULE *;
 9 SYSTEM;
10    A1;
11 PROBLEM;
12    SPC A1 DATION INOUT ALPHIC CONTROL(ALL);
13    SPC P256 ENTRY RETURNS(FIXED) GLOBAL;
14
15
16 PROG: TASK;
17    DCL WORD BIT(16);
18    DCL ADR  FIXED;
19    DCL BYTE FIXED;
20    DCL MODE FIXED;
21    DCL RTN  FIXED;
22
23    PUT 'addr: ' TO A1 BY SKIP,A;
24    GET WORD FROM A1 BY SKIP,B4(4);
25    ADR = TOFIXED(WORD);
26
27    PUT 'byte: ' TO A1 BY SKIP,A;
28    GET WORD FROM A1 BY SKIP,B4(2);
29    BYTE = TOFIXED(WORD);
30
31    PUT 'mode: ' TO A1 BY SKIP,A;
32    GET WORD FROM A1 BY SKIP,B4(1);
33    MODE = TOFIXED(WORD);
34
35    RTN = P256(ADR,BYTE,MODE);
36    PUT TOBIT(RTN) TO A1 BY SKIP,B4(4);
37 END;
38 MODEND;
```

So spricht man die Programmier-Prozedur aus PEARL-Programmen an.

Software-Upgrading

Viele Ideen der Entwickler, vor allem aber auch der Nutzer, konnten mit dem Upgrade 1/87 verwirklicht werden. Wie sich die neue Version im einzelnen von der bisherigen unterscheidet, möchte ich in diesem Beitrag beschreiben, der unsere Serie über die PEARL-Programmierung abschließt. Aus der Vielfältigkeit der Verbesserungen ergibt sich, daß ich diesmal kein größeres Programmbeispiel vorstellen kann. Auch bei Programmen sind die elerlegenden Wollmilchsäue selten.

Wen die EPROM-Bank am Modulport des Atari ST gestört hat, der kann jetzt RTOS-UH PEARL auch in einer Boot-Version auf Diskette erhalten. Diese unterstützt den Betrieb des c't-Userports ebenso wie die bisherige B-Version. In der Disketten-Version legt sich RTOS in den oberen Teil des RAM, ab etwa $D6000. Somit stehen beim ST + noch über 870 K Byte freien Speichers zur Verfügung.

Beim Kaltstart fällt als erstes die neue Systemmeldung auf. Säuberlich geordnet erscheinen jetzt die Überschrift und die Namen der System-Komponenten auf dem Bildschirm, letztere mit den aktuellen Versionsnummern. Schon an diesen kann man erkennen, daß sich zwischenzeitlich so einiges getan hat.

Die bisherige Beschränkung auf sechs Buchstaben für Namen kann jetzt niemand mehr als Ausrede für unverständliche Abkürzungen heranziehen. Künftig kann der Programmierer bis zu 24 Buchstaben zu einem bezeichnenden Namen nutzen. (Ausnahme: Dateinamen dürfen weiterhin nur acht Zeichen lang sein.) Diese Änderung kann Probleme nach sich ziehen bei Programmen, die davon ausgehen, daß Namen nicht länger als sechs Buchstaben sind. Diese müssen entsprechend geändert und neu übersetzt werden.

Ein wenig in Richtung MSDOS und CP/M geht die Änderung des Trennzeichens hinter Datenstationsnamen: hier ist der Punkt dem Doppelpunkt gewichen. Nur in der Dation-Spezifikation im Systemteil von PEARL-Programmen wird auch weiterhin der Dateiname durch einen Punkt von dem Namen der Datenstation getrennt.

Im diesem Zusammenhang möchte ich auf die Datenstation 'UL:' hinweisen, die ein wenig ein Schattendasein führt. Sie ist in der Tabelle der Stationsnamen in der Spalte der ACIA 2 zu finden. Diese Datenstation ist für die Kommunikation mit anderen Systemen gedacht, die zur Satztrennung nicht ein einfaches CR, sondern CR-LF verwenden. Wenn man von der seriellen Schnittstelle ein derartiges File in ein ED-File kopiert, ist jede zweite Zeile eine Leerzeile. Um diese störenden LFs zu eliminieren, kann man die Station 'UL:' benutzen, etwa

```
COPY UL:>ED.SI
```

Ansonsten verhält sich 'UL:' exakt wie die Datenstation B2.

Pipes mit Namen

Vielfältig ließ sich auch bis jetzt schon die Daten-Pipeline VI/VO nutzen. Jetzt kann man für diese Datenstation zusätzlich Namen angeben und damit beliebig viele, voneinander getrennte Datenkanäle zur Kommunikation zwischen Programmen einrichten. Will man etwa ein Programm aus mehreren Modulen von der Diskette direkt in den Speicher laden, so ist dies jetzt mit

```
P F0.MODULE1>VO.P1
P F0.MODULE2>VO.P2
P F0.MODULE3>VO.P3
LOAD VI.P1+VI.P3
LOAD VI.P2
```

möglich – ohne die Gefahr, daß Daten irgendwie gemischt werden. Sind noch Daten in einem Datenkanal, die nicht mehr gebraucht werden, so lassen sich diese mit

```
RM VO.MIST
```

entfernen. RM ist ein neuer Befehl des Kommandointerpreters und steht als Kürzel für Re-Move. Genau wie der Befehl gleichen Namens unter MSDOS dient er zum Löschen von Dateien oder, wie erwähnt, zum Schließen von Datenkanälen.

Time-Sharing

Ein gänzlich anderes Systemverhalten läßt sich durch den Befehl SHARE PRIO xyz erreichen. Normalerweise bearbeitet RTOS immer die Task höchster Priorität, solange, bis sie fertig oder nicht mehr lauffähig ist. Haben Tasks dieselbe Priorität, so wird zuerst diejenige abgearbeitet, die in der Liste vorne steht. Ist diese erste Task in der Liste aber ein 'Dauerrenner', so kommen die übrigen nicht zum Zuge.

Hier setzt nun das SHARE-Kommando an. Es erzeugt eine Subtask, die sich als erstes Glied einer Prioritätsebene einbaut und die auf diese Priorität entfallende Prozessorkapazität an die nachfolgenden Tasks verteilt. Damit sich SHARE an die erste Stelle in einer Prioritätsebene 'vordrängeln' kann, muß die beim Aufruf angegebene Priorität um eins höher sein als die der Prioritätsebene, in der im 'Time-Sharing'-Verfahren gearbeitet werden soll. Will man also etwa mehrere Compiler parallel arbeiten lassen, die normalerweise die Priorität 20 haben, so ist SHARE PRIO 19 aufzurufen. Mit einem UNLOAD SHRDxx läßt sich wieder auf rein prioritätengesteuertes Verhalten zurückschalten.

Erweitert worden ist auch der S-Befehl, mit dem man sich die Speicherbelegung ansehen kann. Da die verfügbaren Speicher und mit ihnen die Dateien immer größer wurden, faßt der S-Befehl jetzt aufeinanderfolgende ED-File-Blöcke, die zu einer Datei gehören, zusammen und kennzeichnet dies durch ein '+' hinter der zweiten Adresse. Will man alle Blöcke angezeigt bekommen, kann man dies durch den Befehl 'S A' erreichen.

Eine Besonderheit der Atari-ST-Implementierung, der Befehl HCPY zum Drucken einer Bildschirmgrafik, ist jetzt mit einer Schnittstelle versehen worden. Wer einen Matrixdrucker hat, der nicht IBM-kompatibel ist, kann sich jetzt seinen eigenen Treiber schreiben und an diesen Befehl anbinden.

Mit ä, ö, ü und ß

Mit dieser Revision hat die Terminalemulation die deutschen Umlaute und Sonderzeichen gelernt. Mit 'ESC E' beziehungsweise 'ESC R' läßt sich zwischen deutschem Zeichensatz und US-ASCII hin- und herschalten. Da der Zeichensatz jetzt im RAM liegt, kann man aber auch eigene Zeichensätze laden oder einzelne Zeichen ändern. Bei der Programmierung von Bildschirmmasken dürfte von Nutzen sein, daß der Cursor jetzt

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /*  R T O S / U H   Rev 2.0   D E M O                            */
 4 /*                                                               */
 5 /*  last update:   16.01.87    13h05                              */
 6 /*                                                               */
 7 /*****************************************************************cmt/
 8 MODULE M;
 9 SYSTEM;
10    A1;
11 PROBLEM;
12    SPC A1  DATION INOUT ALFHIC CONTROL(ALL);
13
14    SPC SPRITL ENTRY
15                 (FIXED, FIXED, FIXED, FIXED, CHAR(5), BIT(16)) GLOBAL;
16    SPC SPRITS ENTRY
17                 (FIXED, FIXED, FIXED, FIXED, CHAR(5), BIT(16)) GLOBAL;
18    SPC CIRCLE ENTRY (FIXED, FIXED, FIXED, FIXED) GLOBAL;
19    SPC TEXT   ENTRY (FIXED, FIXED, FIXED, CHAR(80)) GLOBAL;
20    SPC CURPOS ENTRY (CHAR, CHAR) GLOBAL;
21
22    DCL SAVE (7,16)   BIT(16); /* to save picture under text */
23    DCL ESC  CHAR;
24
25
26 /*** CURSET*********************************************************/
27 /*  procedure to position the cursor                               */
28 /*****************************************************************/
29 CURSET: PROC (row FIXED, column FIXED);
30    PUT ESC, '=', TOCHAR(row + 32), TOCHAR(column + 32)
31       TO A1 BY (4)A;
32 END;
33
34
35 /*** DEMO ***********************************************************/
36 /*  task to show the enhanced graphic capabilities                 */
37 /*****************************************************************/
38 DEMO: TASK;
39    DCL (X, Y) FIXED;
40    DCL (CX, CY) FIXED;
41    DCL Colour FIXED;
42
43    CX = 30;
44    CY = 1;
45    PUT ESC,'*' TO A1;                          /* clear the screen */
46    CALL CURSET (CY, CX);                     /* position the cursor */
47    PUT ESC, 'E', ESC, '[8]berschrift' TO A1 BY (4)A; /* underline ..*/
48    PUT ESC, 'G0' TO A1 BY (4)A;                /* reset underline */
49    CY = CY + 2;
50    FOR I TO 9 REPEAT                                          /* draw box */
51       CALL LINE (20 + I, 20, 20 + I, 180, 1);
52       CALL LINE (20, 20 + I, 180, 20 + I, 1);
53    END;
54    CALL CIRCLE (90, 90, 42, 1);                  /* .. and circle */
55    X = 22;
56    Y = 27;
57    FOR I TO 46 REPEAT
58       CALL GETPIX (X, Y, Colour);             /* test colour of pixel */
59       IF Colour EQ 1                         /* if pixel set (b/w), .. */
60          THEN CY = CY + 1;                     /* cursor one line down */
61       FIN;                        /* to save position of pixel on screen */
62       CALL SPRITL (X, Y, 40, 16, 'MAIN', SAVE);
63       CALL TEXT (X, Y, -1, 'hello');          /* write inverting text */
64       CALL CURSET (CY, CX);       /* write cursor position and time */
65       PUT 'X: ', X, '  Y: ', Y ,'  ',NOW TO A1 BY (2)(A,F(3)),A,T(10);
66       AFTER 0.25 SEC RESUME;                     /* wait a moment .. */
67       CALL SPRITS (X, Y, 40, 16, 'MAIN', SAVE);   /* restore screen */
68       X = X + 1;                              /* advance text position */
69       Y = Y + 1;
70    END;
71    CALL TEXT (X, Y, 1, 'hello');                /* write solid text */
72    CALL TEXT (X + 3, Y + 19, 1, 'world');
73 END;
74
75
76 /*** Uhrzeit ********************************************************/
77 /* task to write the system time to the upper right corner of screen */
78 /*****************************************************************/
79 Uhrzeit: TASK PRIO 1;
80    DCL (oldCX, oldCY) CHAR;
81
82    CALL CURPOS (oldCX, oldCY);        /* get current cursor position */
83    CALL CURSET (0, 72);            /* set cursor to upper right corner */
```

absolut positioniert werden kann. Außerdem gibt es eine Einbaufunktion CURPOS, mit der die augenblickliche Cursor-Position abgefragt werden kann. In dem Demo-Modul ist sie in der Task 'Uhrzeit' zu sehen. Damit man direkt mit den erhaltenen Werten den Cursor wieder positionieren kann, werden als Parameter Daten vom Typ CHAR benutzt, die auch schon den Offset von $20 enthalten, wie er für 'ESC = xpos, ypos' gebraucht wird.

'Uhrzeit' ist eine Task, die den Stand der Systemuhr in die obere rechte Ecke des Bildschirms schreibt. Plant man diese Task regelmäßig ein, erhält man immer die aktuelle Uhrzeit. Ein Problem ergibt sich aus der Datenstationsstruktur unter RTOS. Solange auf eine Eingabe gewartet wird, ist keine Ausgabe möglich. Es wäre ja auch nicht schön, wenn einem mitten in eine Eingabe irgendwelche Daten geschrieben werden, die eventuell auch noch das augenblickliche Bild 'wegscrollen'. Also kommt auch 'Uhrzeit' mit ihrer Ausgabe nicht durch, solange auf eine Eingabe gewartet wird. Ein anderes Problem ergibt sich, wenn man den Editor aufruft und 'Uhrzeit' weiterlaufen läßt. Auf dem Bildschirm kann es dann manchmal chaotisch aussehen, wenn beide Tasks am Cursor herumpositionieren. Aber dies sind natürlich typische Probleme, die in einem Multitasking-System stets auftreten werden, wenn man derartige Abläufe unsynchronisiert miteinander konkurrieren läßt.

In dieser kleinen Task ist noch eine weitere Neuerung enthalten, nämlich eine Erweiterung des Sprachumfangs des PEARL-Compilers: NOW ist vom Datentyp CLOCK und liefert die Uhrzeit des Systems. Dank dieser Erweiterung

```
84    PUT NOW TO A1 BY T(8);                    /* display current time */
85    PUT ESC, '=', oldCX, oldCY TO A1 BY (4)A;      /* restore cursor */
86 END;
87
88
89 /*** START ***********************************************************/
90 /*  task to initialize the stuff of this funny module                */
91 /*********************************************************************/
92 START: TASK;
93    ESC = TOCHAR(27);
94    ALL 0.5 SEC ACTIVATE Uhrzeit;
95    ACTIVATE DEMO;
96 END;
97
98 MODEND;
```

Dieses kleine Demoprogramm führt die verbesserten Grafik- und Terminalfunktionen vor.

konnte die frühere Pseudodatenstation 'CL' entfallen.

Mehr Grafik unter RTOS

Die Task DEMO stellt die verbesserten und zum Teil neuen Grafikfunktionen vor. Die Funktion LINE ist schon bekannt, wurde aber wesentlich schneller. Außerdem kann hier, wie bei den Funktionen SETPIX, CIRCLE und TEXT, ein Invertieren der Pixels erreicht werden, indem man den Farb-Parameter mit negativem Vorzeichen angibt. Ein zweimaliger Aufruf bei gleichen Koordinaten erzeugt also wieder das ursprüngliche Bild.

CIRCLE ermöglicht es, sehr schnell Vollkreise zu zeichnen. Dabei können Mittelpunkt, Radius und Farbe angegeben werden. Ist der Kreis nicht ganz auf dem Bildschirm darzustellen, so wird dies von CIRCLE abgefangen. Man braucht also keine Sorge zu haben, daß irgendwelche Speicherbereiche überschrieben werden.

SPRITL und SPRITS sind RTOS-Anwendern dem Namen nach schon bekannt, haben aber ihr Verhalten ein wenig geändert. Während in der alten Version in dem BIT(16)-Feld immer mit einer Wortgrenze beginnend abgespeichert wurde (auch wenn die Position xpos nicht auf einer Wortgrenze lag), werden jetzt die Bits geschoben, bis das Bit der Position xpos als MSB zu liegen kommt. Das ist natürlich zeitaufwendig und hat diese Funktion verlangsamt. Dafür kann man aber jetzt Bitmuster über den Bildschirm bewegen, ohne sich mit der aktuellen Position innerhalb eines Wortes auseinanderzusetzen. Richtet man es ein, daß xpos mit einer Wortgrenze zusammenfällt, ist auch das alte Tempo wieder da. Mit TEXT kann man auch auf Nebenbildschirmen, wie man sie auf dem Atari unter RTOS-UH einrichten kann, Texte ausgeben. Diese Texte lassen sich pixelgenau positionieren, sind damit aber auch entsprechend langsamer als die Textausgabe auf dem Hauptbildschirm. Ein auszugebender String kann bis zu 80 Zeichen lang sein. Anschließende Leerzeichen werden nicht mit ausgegeben. Wie bei der Funktion CIRCLE wird auch hier automatisch die Bildschirmgrenze überwacht und ein Überschreiben anderer Speicherbereiche verhindert.

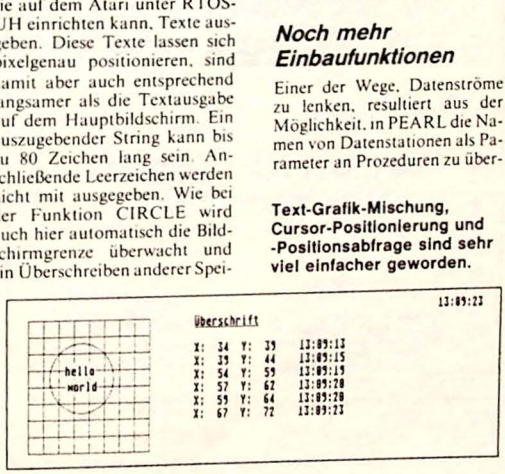
Verbleibt noch die Funktion GETPIX, das Gegenstück zu SETPIX. Mit ihr läßt sich die Farbe eines Bildschirmpunktes ermitteln. Dazu wird eine Variable 'color' übergeben, in die die Funktion die vorgefundene Farbe hineinschreibt. In der Task DEMO benutze ich diese Funktion, um jedesmal, wenn der wandernde Text mit seinem Bezugspunkt links oben eine der Linien kreuzt, diese Position auf dem Bildschirm zu speichern. Dies geschieht durch das Inkrementieren der Cursor-Positionszeile, das nachfolgende Positionsausgaben eine Zeile tiefer erscheinen läßt.

Das Resultat der Task DEMO ist in dem Bildschirmausdruck zu sehen. Die von der Task 'Uhrzeit' stammende Zeitangabe steht oben rechts. Deutlich werden hier auch noch einmal zwei der erweiterten Fähigkeiten der Terminalemulation, nämlich deutscher Zeichensatz und die Möglichkeit, Texte unterstrichen auszugeben.

Noch mehr Einbaufunktionen

Einer der Wege, Datenströme zu lenken, resultiert aus der Möglichkeit, in PEARL die Namen von Datenstationen als Parameter an Prozeduren zu über-

Text-Grafik-Mischung, Cursor-Positionierung und -Positionsabfrage sind sehr viel einfacher geworden.



geben. Wie dies genau abläuft, habe ich bereits in Teil 4 dieser Serie beschrieben. Einen anderen Weg, bei dem nicht schon bei der Programmentwicklung die Datenstationen des Zielsystems bekannt sein müssen, stellt die Funktion ASSIGN dar. Damit kann der Datenstrom einer spezifizierten Datenstation auf eine namentlich zu benennende umgeleitet werden. Dazu werden ihr die Adresse (IDENT) der Datenstation, deren Daten umzulenken sind, sowie ein String, der den neuen Stationsnamen enthält, übergeben. In Verbindung mit der Möglichkeit, beim Öffnen einen File-Namen angeben zu können, lassen sich somit nahezu alle Datenwege erreichen. Zu beachten ist, daß eine Datenumlenkung, die mit ASSIGN durchgeführt wurde, überall innerhalb eines Moduls gültig ist. Wurde die Datenstation global vereinbart, reicht die Wirkung sogar über das eigene Modul hinaus.

ASSIGN und TASKST sind zwei der neuen Einbaufunktionen.

Ein Hilfsmittel ganz anderer Art stellt die Funktion TASKST dar. Mit ihr läßt sich der augenblickliche Status einer Task feststellen. Dazu wird der Name der Task in einer CHAR(24)-Variablen übergeben. Als Resultat bekommt man einen BIT(32)-Wert. Erhält man 'FFFFFFFF' als Resultat, so existiert keine Task dieses Namens. Ansonsten sagt das MSB (Bit 31) aus, ob die Task in der Dispatcher-Queue steht und somit aktiviert wurde. Dieses Bit wird '1', wenn die Task in den Zustand 'DORM', also schlafend, geht. Die Bits 0 bis 7 enthalten die Information, ob die betreffende Task irgendwie eingeplant ist. Sind sie alle '0', so liegt keine Einplanung vor. Die Bits 8 bis 15 sagen schließlich aus, warum eine Task eventuell nicht lauffähig ist. Dies ist zum Beispiel der Fall, wenn sie suspendiert ist oder auf eine langsame Ein-/Ausgabe oder auf die Zuteilung von Workspace durch das Betriebssystem wartet. Gänzlich unproblematisch ist auch diese Funktion nicht. Während eine Task nämlich noch das Resultat auswertet, kann natürlich die untersuchte Task zwischenzeitlich längst ihren Status wieder geändert haben. Die Situation ist vergleichbar mit der beim L-Befehl des Kommandointerpreters. Auch hier wird eine Aussage gemacht, die eventuell schon nicht mehr stimmt.

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /* D E M O  neue Einbaufunktionen   ASSIGN  und  TASKST          */
 4 /*                                                               */
 5 /*****************************************************************/
 6 MODULE X;
 7 SYSTEM;
 8    A1;
 9    OldDation: ED.LONGDUMMYNAME;
10 PROBLEM;
11    SPC A1          DATION INOUT ALPHIC CONTROL(ALL);
12    SPC OldDation DATION INOUT ALPHIC CONTROL(ALL);
13
14    SPC TASKST ENTRY (CHAR(24)) RETURNS (BIT(32)) GLOBAL;
15    SPC ASSIGN ENTRY (DATION INOUT ALPHIC CONTROL(ALL) IDENT,
16                     CHAR(24)) GLOBAL;
17
18
19 Test1: TASK;
20    DCL DestDation CHAR(24);
21    DCL DestFile   CHAR(24);
22
23
24    GET DestDation FROM A1 BY SKIP,A;       /* get Dationname  */
25    GET DestFile   FROM A1 BY SKIP,A;       /*  and Filename   */
26
27    CALL ASSIGN(OldDation, DestDation);     /* set Dation      */
28    OPEN OldDation BY IDF(DestFile);        /*  and Filename   */
29
30 END;  /* Test1: TASK */
31
32
33 Test2: TASK;
34    DCL TaskName   CHAR(24);
35    DCL Status     BIT(32);
36
37    TaskName = 'AnyTask';
38    Status = TASKST(TaskName);
39    WHILE NOT Status.BIT(32) REPEAT    /* wait until task finished */
40       AFTER .2 SEC RESUME;
41       Status = TASKST(TaskName);
42    END;
43
44 END;  /* Test2: TASK */
45
46 MODEND;
```

Aus Full-PEARL stammt der jetzt ebenfalls verfügbare Operator REM. Mit ihm läßt sich der Rest einer Ganzzahldivision bestimmen. So ergibt 15 REM 4 als Resultat den Wert 3.

REFs – nicht ganz ungefährlich

Ebenso aus Full-PEARL stammt die REF-Variable. Hinter dieser Bezeichnung verbirgt sich nichts anderes als ein Pointer. Ein solcher Zeiger kann auf die Grunddatentypen sowie auf Strukturen zeigen, nicht aber auf ganze Felder – einzelne Feldelemente wiederum sind zulässig.

Die Verwendung von Zeigern ist grundsätzlich nicht unproblematisch. Der 'Vater' des

RTOS-UH/PEARL-Systems, Professor Dr.-Ing. Wilfried Gerth, schreibt dazu in dem neuen Handbuch: 'Wer mit REF-Variablen arbeitet, begibt sich in einen Raum, in dem unser Compiler keinen Schutz vor Unfug mehr geben kann! Eine Zeigervariable kann wer weiß wohin zeigen – und damit kann wer weiß was im System zerstört werden. Gleichwohl sind aber mit REFs Konstrukte programmierbar, die ihren (gefährlichen) Einsatz rechtfertigen.'

Deklariert wird ein Zeiger wie eine normale Variable mit dem Unterschied, daß statt 'Datentyp' jetzt 'REF Datentyp' steht. Auch REF-Variable lassen sich bei der Deklarierung initialisieren (REF-Syntax, Zeile 6). Genau zu unterscheiden ist stets zwischen dem Setzen eines Zeigers und der Zuweisung eines Wertes an die Variable, auf die der Zeiger zeigt. Im letzteren Fall wird auf der linken Seite der Zuweisung der Operator CONT verwendet. Da damit eindeutig ist, daß es sich nicht um das Setzen eines Zeigers handelt, steht auf der rechten Seite kein CONT. So wird in dem Listing DEMO REF-Variable in Zeile 65 der Zeiger Ptr gesetzt, während in Zeile 68 der Variablen, auf die Ptr zeigt, der Inhalt der Struktur Work zugewiesen wird.

In dem Programm DEMO REF initialisiere ich in der Task TEST durch Zeiger verkettete Strukturen, die jeweils einen String, seine Position auf dem Bildschirm sowie ein Attribut enthalten. Die Prozedur DrawTree gibt die Strings in der Kette ab einem wählbaren Glied an der angegebenen Stelle auf dem Bildschirm aus, bis das Ende der Kette gefunden wird.

Zeigt eine REF-Variable auf eine Struktur, wird die gleiche Syntax verwendet wie bei der

```
 1 /*****************************************************************/
 2 /*   S Y N T A X :   R E F - V a r i a b l e                     */
 3 /*****************************************************************cem*/
 4
 5 DCL Zahl    FIXED;
 6 DCL Zeiger  REF FIXED INIT(Zahl); /* Deklaration und Initialisierung */
 7
 8
 9 Zeiger = Zahl;                  /* Zeiger setzen */
10 CONT Zeiger = Zahl;             /* der Variablen, auf die Zeiger zeigt, */
11                                 /*  den Inhalt von Zahl zuweisen        */
12 IF Zeiger IS Zahl THEN  ...     /* Zeiger zeigt auf Zahl                */
13
14 IF Zeiger1 ISNT Zeiger2
15    THEN  ...                    /* Zeiger zeigen auf verschiedene Variable */
16
```

```
 1 /*****************************************************************/
 2 /*                                                               */
 3 /*   D E M O   R E F - V a r i a b l e                           */
 4 /*                                                               */
 5 /*****************************************************************cem*/
 6 MODULE M;
 7 SYSTEM;
 8    A1;
 9 PROBLEM;
10    SPC A1        DATION INOUT ALPHIC CONTROL(ALL);
11
12    TYPE OBJECT STRUCT */
13        String        CHAR(10),
14        CX            FIXED,
15        CY            FIXED,
16        Selected      BIT,
17        Forl          REF OBJECT
18        /);
19
20    DCL Tree(10)  OBJECT;
21    DCL ESC       CHAR;
22
23
24
25 DrawTree: PROC (Startobj OBJECT);
26    DCL Ptr    REF OBJECT;
27    DCL Done   BIT;
28
29    Done = '0'B;
30    Ptr = Startobj;
31    WHILE NOT Done REPEAT
32       IF Ptr.Selected
33          THEN PUT ESC,'G4' TO A1;        /* text attributes inverse */
34       FIN;
35       PUT ESC, '=', TOCHAR(Ptr.CY + 32), TOCHAR(Ptr.CX + 32)
36          TO A1 BY (4)A;
37       PUT Ptr.String, ESC, 'G0' TO A1;
38       IF Ptr.Forl IS NIL
39          THEN Done = '1'B;               /* it's done .. */
40          ELSE Ptr = Ptr.Forl;
41       FIN;
42    END;
43 END;  /* DrawTree: PROC */
44
45
46
47
48 TEST: TASK;
49    DCL C    CHAR;
50    DCL Work OBJECT;
51    DCL Ptr  REF OBJECT;
52
53
54    ESC = TOCHAR(27);
55    Ptr = Tree(1);
56    TO 10 REPEAT
57       Ptr.Forl = NIL;
58       CALL REFADD (Ptr, 1);
59    END;
60
```

```
61     Ptr = Tree(1);
62     Ptr.Forl = Tree(4);          Ptr.String = 'ROOT';
63     Ptr.CY = 10;  Ptr.CX = 20;   Ptr.Selected = '1'B;
64
65     Ptr = Tree(4);
66     Work.Forl = Tree(2);         Work.String = 'Hello.';
67     Work.CY = 9;  Work.CX = 35;  Work.Selected = '0'B;
68     CONT Ptr = Work;
69
70     Ptr = Tree(2);
71     Ptr.Forl = NIL;              Ptr.String = 'Friends';
72     Ptr.CY = 11;  Ptr.CX = 40;   Ptr.Selected = '0'B;
73
74     PUT ESC,'*' TO A1;           /* clear entire screen */
75     CALL Draw-Tree(Tree(4));
76     GET C FROM A1 BY 9*IP,A;
77     PUT ESC,'*' TO A1;           /* clear entire screen */
78     CALL Draw-Tree(Tree(1));
79
80 END;  /* TEST: TASK */
81
82
83 MODEND;
```

REF-Variablen – ein weiterer Schritt in Richtung Full-PEARL.

Struktur-Variablen selbst, um auf die einzelnen Elemente zuzugreifen (Zeile 35 bis 37).

Als Inhalt für nicht gesetzte Zeiger gibt es NIL. Mit diesem Wert initialisiere ich die Zeiger in dem Feld Tree in Zeile 56 bis 59. Dies wird später zur Erkennung des Endes der Zeigerkette gebraucht. Die Einbaufunktion REFADD, die ich dabei benutze, setzt den Zeiger um jeweils ein Datenobjekt weiter. Dabei muß sich der Programmierer nicht um die Frage kümmern, wie viele Bytes das einzelne Datenobjekt lang ist. Um so mehr muß er aufpassen, daß mit REFADD der Zeiger nicht das Feld verläßt und das Betriebssystem gefährdet. Diese Initialisierung hätte natürlich auch mit einer benamten Schleifenvariable als Index durchgeführt werden können. Mit der REF-Variablen ist man aber schneller, da REFADD kein Funktionsaufruf ist, sondern vom Compiler direkt in entsprechenden Maschinencode umgesetzt wird. Dies macht sich vor allem bei größeren und mehrdimensionalen Feldern in einem zeitlichen Faktor von 2 bis 6(!) bemerkbar.

Bleibt noch das Problem, Zeiger zu vergleichen. Zu diesem Zweck gibt es IS und ISNT. Auf diese Weise stelle ich in Zeile 38 fest, ob das Ende der Zeigerkette erreicht ist. Ist es nicht erreicht, wird in Zeile 40 der Zeiger auf das nächste Glied in der Kette gesetzt.

Bei Prozeduraufrufen werden Zeiger generell dereferenziert. Das heißt, daß immer der Inhalt der Variablen, auf die der Zeiger zeigt, übergeben wird. Ein Zeiger kann nur innerhalb einer Struktur an eine Prozedur als Parameter übergeben werden. Die Struktur kann natürlich als alleiniges Element den Zeiger enthalten.

Für 68020 vorbereitet

Mit der neuen Version ergeben sich auch Änderungen für den Assembler-Programmierer. Sie ist nämlich auch für Rechner mit der CPU 68020 vorgesehen und braucht deshalb etwas mehr Platz im Task-Kopf als früher. Dort, wo bisher 40 Bytes Platz freizuhalten waren, müssen es jetzt 64 Bytes sein. Natürlich laufen alte Systeme auch mit diesen längeren Task-Köpfen, nur umgekehrt kann es schnell zu unangenehmen Folgen kommen.

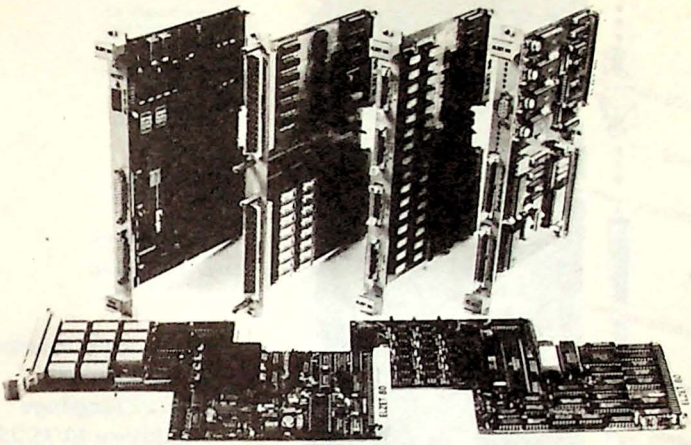
Die jetzt möglichen längeren Namen haben natürlich nicht mehr in dem alten Namensfeld Platz. Um trotzdem möglichst wenig an diesem zu ändern, ist jetzt, wenn man einen Namen mit mehr als sechs Buchstaben benutzen möchte, in dem alten Namensfeld die relative Adresse des Namens anzugeben. Die Unterscheidung zwischen Namen und Adressen erfolgt anhand der Bytes 5 und 6 des Namensfeldes. Sind sie beide 00, so wird angenommen, daß es sich bei den ersten vier Bytes um eine Adresse handelt, sonst werden die sechs Bytes wie bisher interpretiert.

Der neue Compiler ist dafür verantwortlich, daß der Assembler-Programmierer ein Register weniger zur freien Verfügung hat als bisher, und zwar das Register A6. Dies dient jetzt als Zeiger für verschiedene Laufzeitfunktionen und kann nicht mehr frei benutzt werden.

Wer Interrupt-Routinen kodieren möchte, hat künftig zusätzlich die Zelle IID ($7FE) zu versorgen. Genaueres hierzu habe ich in der letzten Folge beschrieben.

Besonders begrüßen werden Assembler-Programmierer die vollständige Liste der Betriebssystem-Traps mit genauer Angabe der Ein- und Ausgabeparameter in dem neuen Handbuch. Damit erübrigt sich so manches Probieren mit letztlich doch unsicherem Ergebnis. PEARL-Programmierer finden im neuen Handbuch die PEARL-Syntax in Backus-Naur-Form vor. Damit läßt sich nach einiger Übung sehr einfach überprüfen, ob Programme zumindest syntaktisch fehlerfrei sind. ct

RTOS-UH/PEARL
auf VMEbus und ECB



Sie kennen jetzt RTOS-UH und PEARL.

Für den industriellen Einsatz von PEARL empfehlen wir Ihnen industrielle Bussysteme.

Auf VMEbus mit der CPU68000 und auf ECB mit 68008.

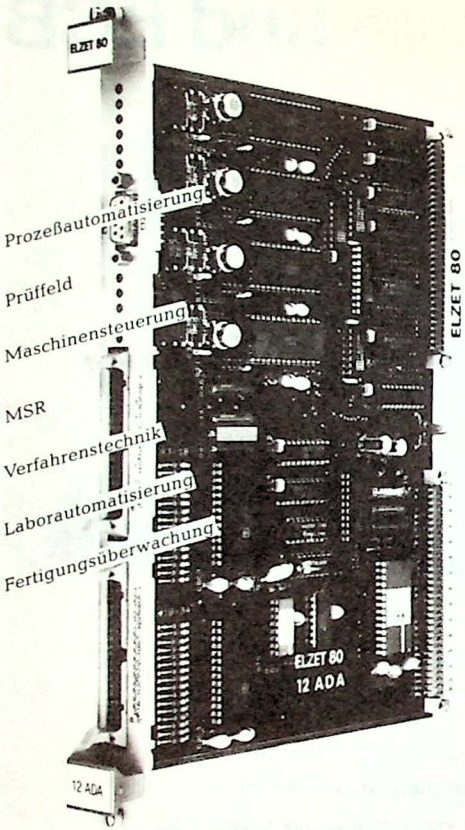
Oder als Einplatinencomputer für ganz kleine Lösungen.

ELZET 80 liefert seit 1978 modulare Mikrocomputersysteme. Weit über 100 Baugruppen für alle Anwendungsbereiche gehören zu unserem Standardprogramm. Für Spezialanwendungen fertigen wir kundenspezifische Lösungen. Vertrauen Sie unserer Erfahrung!

Wir senden Ihnen gerne Kataloge.

ELZET 80
Mikrocomputer GmbH & Co. KG
Wilhelm-Mellies-Straße 88
D-4930 Detmold 18
Tel. 0 52 32 - 81 31 · Tx. 931 473 elzet d

Zum Beispiel: 12 ADA



Zu VMEbus, ECB und RTOS beraten Sie gerne auch unsere Ingenieurbüros (mit Lager):

ESmed GmbH
Fehlerstraße 5
1000 Berlin 41
Tel. 030/851 19 00

MEK GmbH
Düppelstraße 71
2300 Kiel 1
Tel. 04 31/80 42 20

GMS mbH
Marquardstraße 62
6000 Frankfurt 90
Tel. 069/78 87 52

Meditec-Data GmbH
Villastraße 9
7000 Stuttgart 1
Tel. 07 11/28 31 03

PTL GmbH
Schulstraße 28
8000 München 19
Tel. 089/16 99 77

Schweiz:
Bernhard-Elektronik
Aarauer Straße 20
CH-5734 Reinach AG
Tel. 064/71 69 44

- *12-Bit-A/D-Umsetzer*
- *12 Bit Linearität*
- *16 o. 32 Eingänge*
- *Wahlweise 10/15/25 µs max. Umsetzzeit*
- *Eingangsbereiche: 0..10V, ±5V, ±10V*
- *Int. o. ext. Triggerung*
- *S&H mit 3,5 µs*
- *Vier 12-Bit-D/A's*
- *Spannungsausgänge: 0..5V, 0..10V, ±2,5V, ±5V, ±10V*
- *Max. Lin.fehler 1/2 LSB*

Einzelpreis komplett A/D mit 25 µs und 4 Kanäle D/A:

2277,72 DM
(1998,– DM +14 % MWSt.)

ELZET 80
Mikrocomputer GmbH & Co. KG
Wilhelm-Mellies-Straße 88
D-4930 Detmold 18
Tel. 0 52 32 - 81 31 · Tx. 931 473 elzet d

Der Begriff „Multitasking“, EDV-Profis schon lange geläufig, verbreitet sich neuerdings wie ein Lauffeuer auch in der Mikrocomputerszene. Klar, ein moderner Mikroprozessor, vor allem einer aus der 16/32-Bit-Klasse, kann viel mehr leisten, als ein einzelner Benutzer ihm bei Standard-Anwendungen abverlangt. Warum soll die CPU dumm in der Eingabeschleife eines Textprogramms verharren, wenn sie doch gleichzeitig „im Hintergrund“ eine Datei sortieren, eine Mailbox bedienen und die häusliche Heizungsanlage steuern könnte? Unter RTOS-UH/PEARL ist das kein Problem.

In diesem Büchlein geht es um die Programmiersprache, die in idealer Weise die Leistungsfähigkeit des Echtzeit-Multitasking-Betriebssystems RTOS-UH nutzbar macht. PEARL hat sich bei wissenschaftlichen, industriellen und militärischen Großprojekten schon seit langem durchgesetzt. Durch das von Prof. Dr.-Ing. W. Gerth entwickelte integrierte Programmiersystem RTOS-UH/PEARL für 68000-Rechner wurde Echtzeit-Multitasking auf Hochsprachen-Ebene auch für den privaten Anwender leicht erschwinglich.

ISBN 3-88229-o5o-1